Johannes Kepler Universität Linz

Grafisches Werkzeug zur Integration von Data Marts

Diplomarbeit

zur Erlangung des akademischen Grades Mag.rer.soc.oec.

im Diplomstudium Wirtschaftsinformatik

Eingereicht am

Institut für Wirtschaftsinformatik

Data and Knowledge Engineering

Eingereicht von

Lorenz Maislinger

Begutachter

o. Univ. Prof. Dr. Michael Schrefl

Mitbetreuer

Mag. Stefan Berger

Linz, im April 2009

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit mit dem Titel „Grafisches Werkzeug zur Integration von Data Marts“ selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, im April 2009 Lorenz Maislinger

Zusammenfassung

In Zeiten der Globalisierung steigt die Zahl der Unternehmenszusammenschlüsse- und kooperationen stetig an. Dies erfordert oftmals Data Warehouse übergreifende Analysen. Aufgrund einer fehlenden zentralen Instanz zur Verwaltung der Data Warehouses bietet sich die Anwendung eines föderierten Data Warehouses an. Ein föderiertes Data Warehouse stellt dem Anwender mithilfe eines „virtuellen Würfels“ in Form eines globalen Schemas einen transparenten Zugriff auf die verschiedenen Data Warehouses zur Verfügung. Um diesen Zugriff zu ermöglichen müssen die Heterogenitäten zwischen den verschiedenen Data Warehouses beseitigt werden können, hierfür wird die Abfragesprache SQL-MDi eingesetzt.

Ziel dieser Diplomarbeit ist die Umsetzung eines Werkzeuges zur Integration von unterschiedlichen Data Marts und zur Erstellung eines globalen Schemas. Demnach stellt das Werkzeug einen Teil eines föderierten Datawarehouse-Systems dar. Grundlegende Aufgabe des Werkzeuges ist es die Daten für den Abfrageprozess bereitzustellen. Hierzu müssen die Heterogenitäten zwischen den verschiedenen importierten Data Marts beseitigt werden. Für den Abfrageprozess werden der SQL-MDi Query Parser sowie der SQL-MDi Query Processor eingesetzt. Die Umsetzung des Werkzeuges erfolgte als Plug-in für die Rich-Client-Plattform Eclipse. Darüberhinaus wurden von Eclipse bereitgestellte Frameworks, wie z.B. EMF und GEF, eingesetzt. Dies führt zu einem teilweise modellbasierten Ansatz bei der Entwicklung des Werkzeuges. Zusätzlich wurde ein spezielles Augenmerk auf eine benutzerfreundliche Gestaltung des Werkzeuges gelegt.

Abstract

In times of globalization a constant increase in the number of mergers and acquisitions among companies can be observed. This leads to a growing demand on cross-Data Warehouse analysis. Due to the fact that there is no central authority for managing Data Warehouses, the application of federate Data Warehouses is suitable. A federate Data Warehouse uses a "virtual Cube" in form of a global schema to allow transparent access to the various Data Warehouses. A prerequisite for this cross-Data Warehouse access is the removal of existing heterogeneities. In order to achieve this, the query language SQL-MDi is used.

The goal of this thesis is implementing a tool for integrating data marts and designing a global schema. Thus the tool is a part of a federated Data Warehouse System. Its basic task is to provide the data needed to execute the query process of the SQL-MDi Query Parser and the SQL-MDi Query Processor. Hence the heterogeneities among the various data marts have to be eliminated. The tool was implemented as an Eclipse Plug-in. Furthermore the standardized frameworks of eclipse, like EMF or GEF, were used for the implementation of the tool. The usage of EMF induces a model driven architecture approach. Special attention was given to usability of the graphical user interface.

Inhaltsverzeichnis

Abkürzungsverzeichnis

bzw.	beziehungsweise
CWM	Common Warehouse Metamodel
CPL	Common Public License
CORBA	Common Object Request Broker Architecture
DW	Data Warehouse
EMG	Eclipse Modelling Framework
EPL	Eclipse Public License
d.h.	das heißt
GMF	Graphical Modeling Framework
GUI	Graphical User Interface
JDBC	Java Database Connection
JVM	Java Virtual Machine
MDA	Model Driven Architecture
ME/R	Multidimensionale Erweiterung des Entity-Relationship-Modells
MOF	Meta Object Facility
MOLAP	Multidimensional Online Analytical Processing
MVC	Model View Controller
OCL	Object Constraint Language
OLAP	Online Analytical Processing
OMG	Object Management Group
OSGi	Open Service Gateway initiative
ROLAP	Relational Online Analytical Processing
SQL	Structured Query Language
SQL-MDi	Structured Query Language for Multidimensional Integration
SWT	Standard Widget Toolkit
u.a.	unter anderem
xPath	Xml Path
XMI	XML Metadata Interchange
z.B.	zum Beispiel

1 Einleitung

Neben den herkömmlichen Produktionsfaktoren wie Boden, Arbeit und Kapital gewinnt in der heutigen Gesellschaft der Faktor Information immer mehr an Bedeutung. Zur Speicherung, Abfrage und Verwaltung von Daten werden meist Datenbanken verwendet [Bauer & Günzel 2004, S. 6].

Die in einer Datenbank gespeicherte Information ist oft nur schwer und aufwendig abzurufen. Daraus ergab sich der Wunsch der Wirtschaft nach einer performanten Lösung für die analytische Abfrage von Information [Bauer & Günzel 2004, S. 6]. Eine der erfolgreichsten Ansätze für die Lösung dieses Problems ist das Data Warehouse (DW). Ein DW speichert Daten aus unterschiedlichen operativen Datenbanken auf strukturierte Weise, wodurch eine schnelle analytische Auswertung (siehe Definition 2-2) ermöglicht wird. Um Einschränkungen bzw. spezielle Abfragen für Benutzergruppen zu ermöglichen, werden basierend auf DWs Data Marts (siehe Definition 2-4) erstellt. Data Marts stellen dem Anwender eine Teilmenger der Daten zur Verfügung.

Aufgrund der steigenden Nachfrage nach Information stieg auch die Zahl der DWs in den Unternehmen. Oft wurde bei der Erstellung keine Rücksicht auf Standards oder andere im Betrieb befindliche DWs genommen, wodurch unterschiedliche Strukturen in den DWs entstanden. Aufgrund von z.B. Unternehmenszusammenschlüssen und -kooperationen wurden oftmals DW übergreifende Abfragen benötigt. Bedingt durch die heterogenen Strukturen war dies jedoch nicht möglich. Aus diesem Grund wurden Verfahren entwickelt um unterschiedliche DWs zu verbinden. Es besteht die Möglichkeit basierend auf den unterschiedlichen DWs ein neues physisches DW zu erstellen. Dies ist jedoch oft mit hohen Kosten, Ressourcen und Aufwand verbunden. Im Gegensatz dazu gibt es die Möglichkeit die DWs auf Ebene der logischen Schemata zu verbinden (siehe Definition 2-14). Diese Systeme werden als Föderierte Data Warehouse Systeme bezeichnet [Berger & Schrefl 2008 S. 1].

Diese Arbeit beschäftigt sich mit dem zuletzt genannten Ansatz. Eine genaue Einführung in DWs und speziell in föderierte DWs wird im folgenden Kapitel gegeben.

Für die Abfrage von Information aus unterschiedlichen DWs gibt es eine Vielzahl von möglichen Szenarios. Eines dieser Szenarios tritt auf wenn zwei Unternehmen fusionieren und so unterschiedlich aufgebaute DWs mit ähnlichen Informationen aufeinandertreffen. Möglicherweise haben beide Unternehmen ein DW, welches ihre Verkaufsdaten beherbergt, jedoch bestehen Differenzen im Aufbau des jeweiligen DWs. Mithilfe des in dieser Arbeit beschrieben Ansatzes können solche Differenzen beseitigt werden.

Abschnitt 1.1 behandelt die Motivation, Ziele und Anforderungen an diese Diplomarbeit und das zu erzeugende Werkzeug. Abschnitt 1.2 beschreibt das Rahmenbeispiel welches zur Veranschaulichung der verschiedenen Arbeitsschritte dient. In Abschnitt 1.3 wird der weitere Aufbau dieser Arbeit besprochen.

1.1 Aufgabenstellung und Zielsetzung

Ziel dieser Diplomarbeit ist die Implementierung eines grafischen Werkzeuges (Global Schema Architect) zur Integration von verschiedenen Data Marts auf Schema-Ebene. Hierzu zählen folgende Teilaufgaben:

- Das Werkzeug erlaubt es ein globales Schema zu erstellen, welches als „virtueller Würfel“ die Grundlage für den SQL-MDi Query Processor und SQL-MDi Query Parser bereitstellt.
- Physischen DW Schemata können durch den Global Schema Architect importiert werden. Darüber hinaus werden vom Programm Vorschläge über die empfohlene Struktur des DW gegeben.
- Durch das Werkzeug wird die Beseitigung der Heterogenitäten zwischen dem importierten lokalen Schema und dem globalen Schema ermöglicht.
- Durch das Werkzeug wird das Metadata-Dictionary und die SQL-MDi Datei für die Interaktion mit dem SQL-MDi Query Parser und dem SQL-MDi Query Processor bereitgestellt (siehe Abbildung 1-1).

Grundlegende Voraussetzung für die Umsetzung dieses Werkzeuges ist, dass die verschiedenen Data Marts eine inhaltliche Überlappung beinhalten. Es können somit nur ähnliche Daten

verglichen bzw. zusammengeführt werden. Eine Entscheidung über die Realisierbarkeit eines föderierten DW muss in jedem Fall wieder neu vom Benutzer getroffen werden.

Als weitere Grundanforderung an die Umsetzung des Werkzeuges gilt, dass die Implementierung in Form eines Plug-ins auf Basis der Rich-Client-Plattform Eclipse umgesetzt werden muss. Eclipse soll aufgrund der hohen Verbreitung den Wiedererkennungswert beim Benutzer erhöhen und so ein schnelleres Zurechtfinden im Programm ermöglichen. Durch die Verwendung der Standards und Frameworks von Eclipse soll die Kompatibilität zu anderen Werkzeugen ermöglicht werden.

Abbildung 1-1 beschreibt auf einer abstrakten Ebene den grundlegenden Kontext zur Verwendung des Werkzeuges. Durch einen „virtuellen Würfel" wird ein transparenter Zugriff auf verschiedene Data Marts gegeben. Für den Benutzer ist somit nicht wahrnehmbar, dass die gestellte Abfrage auf mehrere Data Marts verteilt wird. Hierzu wird ein globales Schema („virtueller Würfel") eingesetzt, welches die lokalen Würfel integriert und so dem Benutzer eine einzige Abfragestelle bereitstellt. Aufgrund von Differenzen die zwischen dem globalen Schema und dem jeweiligen lokalen Schemen bestehen können, ist es für eine erfolgreiche Abfrage notwendig diese Differenzen zu beseitigen. Dazu wird die SQL-Erweiterung SQL-MDi verwendet. Diese Sprache kann, durch Festlegen von Mappings, die Differenzen zwischen den verschiedenen Schemata ausgleichen.

In Abbildung 1-1 wird der Prozess der Abfrage über mehrere Data Marts dargestellt. Durch den SQL-MDi Query Parser und den SQL-MDi Query Processor werden die Eingabedaten auf syntaktische Korrektheit überprüft und in eine Baumstruktur zerlegt. Durch die Umstrukturierung wird eine Verteilung der Abfrage auf die verschiedenen Data Marts ermöglicht. Schlussendlich werden die aus den einzelnen Data Marts erhaltenen Ergebnisse auf eine Repräsentation des globalen Schemas zusammengefasst. Die Arbeitsschritte, welche dieses Projekt betreffen, sind in Abbildung 1-1 grau eingefärbt. Eine detaillierte Beschreibung dieses Prozesses findet sich in 6.2.

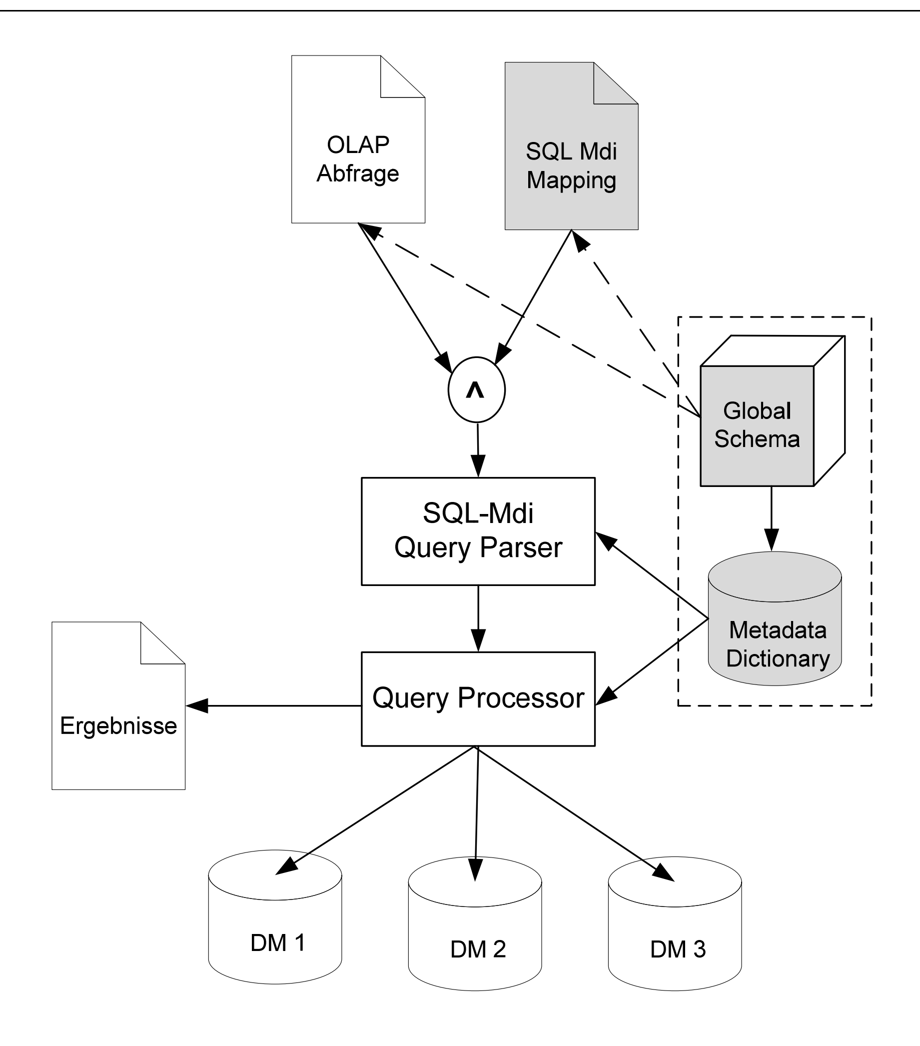


Abbildung 1-1: Definition des kontextuellen Zusammenhangs des Werkzeuges

Neben der praktischen Entwicklung des Prototyps wird in dieser Arbeit in weiterer Folge verwandte wissenschaftliche Literatur aufgearbeitet. Im Speziellen wird auf die Grundlagen von DWs sowie von föderierten DWs eingegangen, um dem Leser eine fundierte Basis für die Umsetzung zu geben. Zusätzlich werden das theoretische Basiswissen zu Eclipse, die von Eclipse bereitgestellten Frameworks sowie die Entwicklung von Plug-ins bereitgestellt.

1.2 Rahmenbeispiel

Als Rahmenbeispiel für dieses Projekt wird auch das von [Brunneder 2008] und von [Rossgatterer 2008] verwendete Beispiel herangezogen. Dadurch wird ein Vergleich von dieser zu den beiden Arbeiten erleichtert. Dies ist von Vorteil da sich die genannten Arbeiten u.a. mit dem SQL-MDi Query Processor und Parser als auch mit der SQL-MDi Sprache beschäftigten.

Das Beispiel behandelt zwei im selben Markt tätige Mobilfunkunternehmen MFA und MFB. MFB hat einen amerikanischen Mutterkonzern und wird durch MFA akquiriert. Da beide Mobilfunkunternehmen zur Unterstützung der Entscheidungsfindung in ihrem jeweiligen Unternehmen ein DW einsetzen ist, es notwendig beide DWs zu integrieren. Da eine physische Integration von zwei unterschiedlichen DWs in ein neues physischen DW sehr viel Zeit und Ressourcen beansprucht wird in diesem Fall der Einsatz eines föderierten DW bevorzugt. Aufgrund der Tatsache, dass beide Unternehmen in einem ähnlichen Markt agieren und deren DW einen ähnlichen Aufbau vorweisen, ist die grundsätzliche Voraussetzung für die Erstellung eines föderierten DW-Systems gegeben. Durch die Einführung eines föderierten DW-Systems besteht die Möglichkeit Daten aus verteilten und unabhängigen DWs auszuwerten ohne die Autonomie der verschiedenen DWs zu beinträchtigen [Rossgatterer 2008 S. 4].

Das in Abbildung 1-2 (a) dargestellte DW von MFA sichert die Kennzahlen der Umsatzzahlen in drei Kenngrößen: Dauer (in Minuten), Umsatz_Telefonie und Umsatz_Sonstiges. Definiert werden diese durch die Dimensionen Kunde, Mobilnetz und Datum. Die Dimension Kunde ermöglicht eine Aggregation nach Sozialversicherungsnummer (svnr) und Tarif. Mittels der Dimension Mobilnetz kann nach verschiedenen Anbietern aggregiert werden. Die Hierarchie der Dimension gliedert sich in Zeit (Minuten), Tag, Monat und Jahr. Als Währung wird in diesem Schema € verwendet. [Rossgatterer 2008 S. 5].

In Abbildung 1-2 (b) wird das DW von MFB dargestellt. Die Umsatzzahlen werden durch die Kenngrößen Gesprächsdauer und Umsatz definiert. Die Kenngrößen werden durch die Dimensionen Kunde, Mobilnetz, Werbeaktion, Datum und Umsatzkategorie beschrieben. Kunde ermöglicht die Analyse nach einzelnen Kunden und Tarifen. Die Aggregation nach Mobilnetz erfolgt nach einzelnen Mobilnetzen. Werbeaktion wird durch die Aggregationsstufe Werbetyp erweitert. Die Hierarchie Datum gliedert sich in Datum, Monat, Quartal und Jahr. Die Dimension Umsatzkategorie ermöglicht die Analyse nach dem Umsatz zugeordnete Umsatzkategorien. Als Währung wird in diesem Schema $ verwendet [Rossgatterer 2008 S. 5].

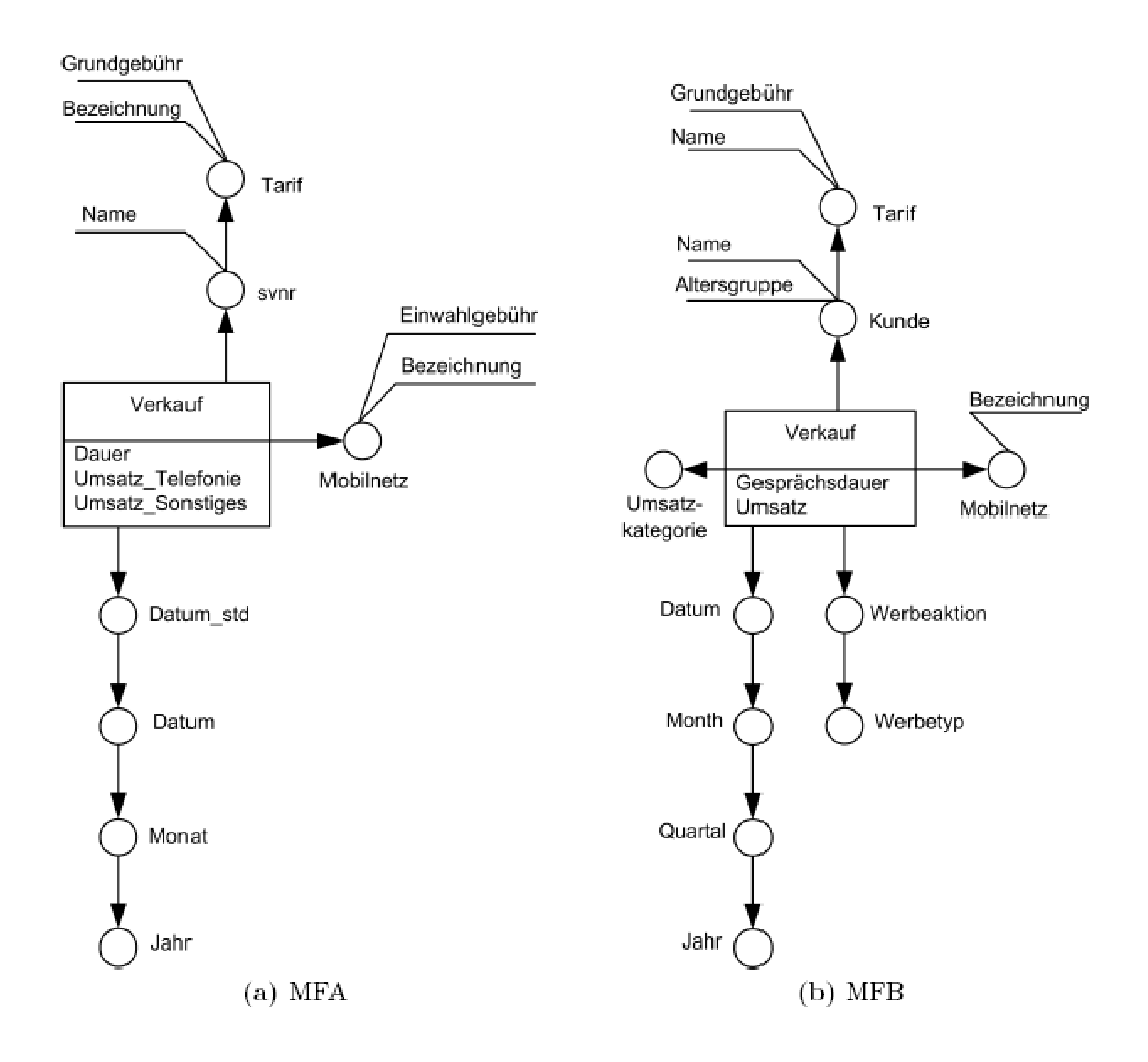


Abbildung 1-2: Würfel für MFA und MFB [Rossgatterer 2008]

1.3 Aufbau der Arbeit

Die Arbeit gliedert sich in sieben verschiedene Kapitel, welche wiederum auf zwei Abschnitte aufgeteilt sind.

In Abschnitt I Grundlagen wird das notwendige Basiswissen für das theoretische als auch praktische Verständnis der in Abschnitt II besprochenen Konzepte und Technologien aufbereitet. Abschnitt II beschäftigt sich mit der Erstellung des grundlegenden Metamodells sowie der tatsächlichen Entwicklung des Protototyps.

Kapitel 2 bespricht die relevanten Grundlagen für DWs und föderierte DWs. Diese sind die Grundlage für die in dieser Diplomarbeit besprochene Thematik. Insbesondere wird auf den Aufbau von föderierten DWs eingegangen. Zusätzlich wird eine kurze Einführung in die SQL-MDi Sprache gegeben, da diese eine Grundlage für das weitere Vorgehen ist.

Die modellgetriebene Architektur wird in der Entwicklung mit Eclipse und dessen Standards oftmals eingesetzt. Aus diesem Grund wird dieser Ansatz, sowie alle für dieses Projekt notwendigen Technologien und Konzepte, in Kapitel 3 beschrieben.

Um die Grundlagen für die Entwicklung des Eclipse Plug-ins festzulegen wird in Kapitel 4 eine kurze Einführung in die Geschichte von Eclipse als auch eine Beschreibung der in dieser Arbeit relevanten Technologien gegeben. Daraufhin werden die Grundlagen für die Erstellung von Metamodellen in Eclipse mit den dazugehörigen Frameworks (z.B. EMF, GEF, SWT) beschrieben.

Kapitel 5 beschäftigt sich mit der Erstellung des Metamodells für das zu erstellende Werkzeug. Dabei werden verschiedene Ansätze aus der Literatur verglichen und auf die Kompatibilität mit dem Eclipse Modell geprüft.

Das darauffolgende Kapitel 6 bespricht die Grundlagen für die Implementierung des Global Schema Architects anhand dem Prozessmodells des evolutionären Prototyps. Nach der Einführung in das Prozessmodell wird auf die an das Projekt gestellten Anforderungen und schließlich auf die Architektur des Werkzeuges eingegangen.

In Kapitel 7 wird die Systemimplementierung basierend auf der in Kapitel 6 definierten Systemarchitektur beschrieben. Im Zuge der Implementierung werden u.a. auch die angewandten Design Patterns detailliert beschrieben.

Die ordnungsgemäße Einbettung des Werkzeuges in das föderierte DW System wird in Kapitel 8 durch einen Integrationstest überprüft. In diesem Zusammenhang wird im Speziellen auf die korrekte Darstellung der Schnittstellen zu den anderen Werkzeugen eingegangen.

Kapitel 9 schließt die Arbeit ab und fasst nochmals die wichtigsten Erkenntnisse zusammen. Zusätzlich wird ein Ausblick auf mögliche zukünftige Erweiterungen und Verbesserungen gegeben.

Um eine kompakte, einheitliche und saubere Trennung der wichtigsten Begriffe dieser Arbeit zu bewerkstelligen, werden in den jeweiligen Kapiteln die grundlegenden Begriffe als Definition gekennzeichnet. Eine Aufstellung aller Definitionen findet sich im Anhang.

I. Grundlagen

2 Föderierte Data Warehouses

In diesem Kapitel werden die theoretischen Grundlagen der bisherigen Forschung zur Umsetzung von DWs beschrieben. Im ersten Teil liegt der Fokus auf den Basiskenntnissen zu DWs. Im zweiten Teil werden föderierte DWs inklusive einer Einführung in verteilte Datenbanksysteme, sowie die Auflösung von Konflikten und Heterogenitäten zwischen verschiedenen Data Marts behandelt.

2.1 Grundlagen von Data Warehousing

Die Verwendung von Daten war über eine lange Zeit speziell durch eine transaktionale Verarbeitung geprägt. So wurden in operativen Anwendungen hauptsächlich kurze Lese- und Schreiboperationen eingesetzt. In der Gegenwart und der riesigen Datenflut die sich in den letzten Jahrzehnten angesammelt hat, steigt die Nachfrage nach einer auswertenden analytischen Verwendung von Daten [Bauer & Günzel 2004 S. 6].

Zur Lösung dieses Problems sind in den letzten Jahren einige neue Begriffe und Konzepte verstärkt in den Mittelpunkt gerückt. In dieser Arbeit werden im Speziellen Data Warehousing und On-Line Analytical Processing (OLAP) behandelt. In der Fachliteratur werden diese Konzepte u.a. unter dem Begriff „Analytische Informationssysteme“ zusammengefasst. Das DW bezeichnet einen unternehmensweiten, entscheidungsorientierten Datenpool welcher für Analyse- und Auswertungstechniken, wie z.B. OLAP herangezogen werden kann. Abbildung 2-1 veranschaulicht den betriebsübergreifenden Einsatz von analytischen Informationssystemen [Chamoni & Gluchowsk 1998 S. 10ff].

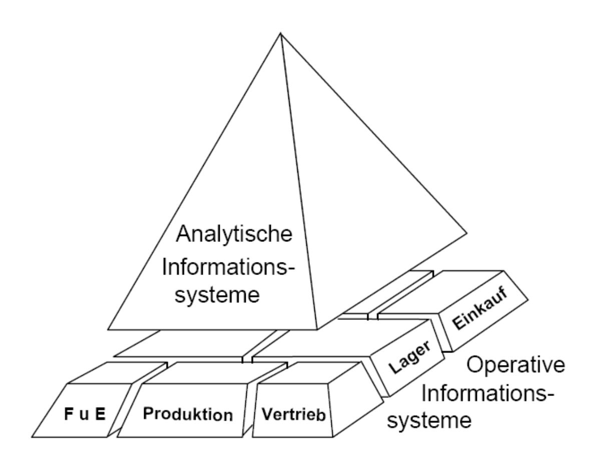


Abbildung 2-1: Analytisches Informationssystem [Chamoni & Gluchowsk 1998 S. 11]

Wie Abbildung 2-1 zeigt, ermöglicht ein analytisches System bzw. ein DW auch abteilungsübergreifende Abfragen über verschiedene operative Datenbanken. Ein DW fasst demzufolge in der Regel mehrere operative Datenbanken zusammen und ist für analytische Abfragen optimiert. Besonders für die Benutzer stellt dies viele Möglichkeiten bereit, für die es jedoch oft notwendig ist über den sprichwörtlichen Tellerrand zu blicken. So bietet ein DW dem Manager z.B. die Möglichkeit Abfragen über mehrere betriebliche Bereiche zu stellen und dadurch Information, die zuvor unerreichbar schien, zu akquirieren [Gardner 1998 S. 54ff].

Definition 2-1: Ein *analytisches System* definiert sich als Datenbank mit dem Ziel Informationen aus aufwändigen Lesetransaktionen, welche sich aus komplexen Abfragen ergeben, zu erhalten. In der Regel besteht ein analytisches System aus einem Zusammenschluss von verschiedenen operativen Datenbanken [Bauer & Günzel 2004 S. 8].

2.1.1 Abgrenzung des Begriffs Data Warehousing

Definition 2-2: Ein Data Warehouse wird als eine fachorientierte, integrierte, nicht-flüchtige und historische Ansammlung von Daten zur Unterstützung von Geschäftsentscheidungen angesehen. Speziell gilt es hervor zu heben, dass das DW ausschließlich analytischen Charakter hat und losgelöst von den operativen Datenbanken eingesetzt wird. („*A data warehouse is a subject oriented, integrated, non-volatile, and time variant collection of data in support of management decisions*“) [Inmon 1996 S. 33]

Aus dieser Definition lassen sich [Bauer & Günzel 2004 S. 7, Heinrich et al. 2004 S. 164] zufolge vier Hauptmerkmale eines DWs formulieren:

Fachorientierung (engl. subject orientation)
Die Darstellung und Speicherung der Daten wird nicht mehr anwendungsbezogen (relational) durchgeführt, sondern am Benutzerziel bzw. am spezifischen Anwendungsziel orientiert. Die Konzentration fällt beispielsweise auf Themenschwerpunkte wie Produkte, Kunden und Verkäufe. Dazu werden die Daten in einem multidimensionalen Schema gespeichert.

Integration (engl. integration)
Die Daten für die weiteren Abfragen des DWs stammen aus mehreren unterschiedlichen integrierten Datenbanken und werden im DW zu einer einheitlichen, integrierten Struktur zusammengefasst. Dadurch soll selbst bei hochgradig heterogenen Datenquellen ein konsistenter und stimmiger Datenbestand erreicht werden.

Nicht flüchtige Datenbasis (engl. non- volatility)
Die Daten in einem DW sind als stabil anzusehen. Diese werden ausschließlich gelesen und nicht verändert oder gelöscht. Es kommt lediglich zu einer analytischen Betrachtung und zu keinem Einsatz im operativen Tagesgeschäft.

Historische Daten (engl. time variance)
Der Faktor Zeit wird gesondert berücksichtigt. Da keine Daten gelöscht werden können, ist es einfach Analysen über längere Zeiträume zu erstellen und so Zeit- und Trendanalysen zu unterstützen. Festzuhalten gilt es jedoch auch, dass die Daten in einem DW nur ein Schnappschuss zum Zeitpunkt des letzten Imports sind. So können Daten zum Zeitpunkt der Abfrage je nach DW Minuten, Stunden, Tage oder gar Wochen alt sein.

Zur Analyse der Daten eines DWs wird in der Regel der Ansatz des „Online Analytical Processings“ (OLAP) verwendet, welcher die Grundlage für die multidimensionale Sicht der Daten darstellt (siehe Abschnitt 2.1.4). Durch den engen Zusammenhang zwischen DW und OLAP werden diese Begriffe oft synonym verwendet. OLAP steht für komplexe Leseoperationen welche einen dynamischen, flexiblen und interaktiven Zugriff auf mehrere Einträge erfordern. Im Gegensatz dazu steht Online Transaction Processing (OLTP) welches speziell operative Datenbankzugriffe (*Lesen und Schreiben*) behandelt [Bauer & Günzel 2004 S. 97ff].

Als Grundlage für die Bewertung der OLAP-Fähigkeit von Software-Werkzeugen werden die von Edgar F. Codd definierten Regeln (vgl. [Codd et al. 1993]) herangezogen.

2.1.2 Notwendigkeit und Einsatzbereiche

Bei der Erstellung eines DW ist die Unterscheidung zwischen transaktionalen (d.h. operativen) und analytischen Daten und Systemen von essentieller Bedeutung. Operative Daten sind in der Regel direkt mit einer betrieblichen Facheinheit verbunden und dafür zuständig die geschäftsbedingten Prozesse des Betriebes zu unterstützen. Im Gegensatz dazu müssen betriebliche Fragestellungen bearbeitet werden, welche in operativen Systemen zu keiner befriedigenden Lösung kommen, was wiederum den Wunsch nach analytischen Auswertungsmöglichkeiten stärkt [Gardner 1998].

Definition 2-3: *Transaktionale Systeme* definieren sich aus der Verwaltung der Daten für kurze und einfache Transaktionen, wie Lesen, Bearbeiten und Löschen [Bauer & Günzel 2004 S. 9]. Im Kontext dieser Arbeit werden transaktionale Systeme generell als Datenbank verwendet.

Um Unterschiede zwischen transaktionalen und analytischen Systemen darzustellen führt [Bauer & Günzel 2004 S. 9ff] einige Vergleiche zwischen transaktionalen und analytischen Systemen an. In Tabelle 2-1 werden die wichtigsten Kriterien für Anfragen wiedergegeben.

Zusätzlich zu den in Tabelle 2-1 angeführten Abweichungen treten große Unterschiede im Bereich des Datenvolumens auf. Transaktionale Systeme haben in der Regel ein Datenvolumen im Bereich Megabyte bis Gigabyte. Im DW liegt die Datenmenge bei Gigabyte bis Terabyte. Dies ergibt sich vor allem daraus, dass die Daten im DW nicht gelöscht werden und sich so eine große Menge von Daten über einen langen Zeitraum ansammelt. Auch in der Anwenderzahl bestehen nennenswerte Abweichungen. So gibt es bei transaktionalen Systemen normalerweise eine enorme Menge an Benutzern, während analytische Systeme eine geringe Anzahl an Benutzern aufweisen [Bauer & Günzel 2004 S. 11ff].

Anfragen	**transaktional**	**analytisch**
Fokus	Lesen, Schreiben, Modifizieren, Löschen	Lesen, periodisches Hinzufügen
Transaktionsdauer und -typ	kurze Lese-/Schreibtransaktionen	lange Lesetransaktionen
Anfragestruktur	einfach strukturiert	komplex
Datenvolumen einer Anfrage	wenige Datensätze	viele Datensätze
Datenmodell	anfrageflexibles Datenmodell	analysebezogenes Datenmodell
Antwortzeit	ms – s	s - min

Tabelle 2-1: Gegenüberstellung der Anfragecharakteristika von transaktionalen und analytischen Anwendungen [Bauer & Günzel 2004 S. 9]

Wie man in Tabelle 2-1 sehen kann sind die Unterschiede zwischen den verschiedenen Verfahren sehr gravierend. Ob der Einsatz eines analytischen Informationssystems in der Praxis gerechtfertigt ist hängt oftmals von unternehmerischen und wirtschaftlichen Entscheidungen ab. Deshalb ist es speziell für kleinere Unternehmen oftmals nicht notwendig ein DW aufzubauen, da die vorhandene Datenmenge diesen Schritt nicht rechtfertigen würde.

Grundlegendes Einsatzgebiet für jedes DW ist die Informationsfindung. Es unterscheiden sich jedoch die Voraussetzungen unter welchen Gesichtspunkten Information abgefragt werden soll. [Bauer & Günzel 2004 S. 11ff] differenziert folgende Anwendungsbereiche:

- Informationsbereitstellung,
- Analyse,
- Planung,
- und Kampagnenmanagement.

Aus den angeführten Anwendungsbereichen ergeben sich einige Einsatzgebiete. Das größte Gebiet stellt das betriebswirtschaftliche dar. Für jegliche betriebliche Prozesse ist genaue und ganzheitliche Information von Interesse. DWs finden auch im Bereich der Wissenschaft Anklang. Sie werden z.B. zur Auswertung von geologischen Messungen verwendet. Selbst technische Arbeiten können durch den Einsatz eines DWs unterstützt werden. Grundlegend gilt es zu sagen, dass die Einführung eines DWs als zweckmäßig und zielführend betrachtet wird,

wenn eine große bzw. unüberschaubare Menge an Daten vorhanden ist und eine analytische Auswertung dieser einen Nutzen bringt.

2.1.3 Architektur eines Data Warehouses

Die Architektur eines DWs stellt sich üblicherweise wie in Abbildung 2-2 dar.

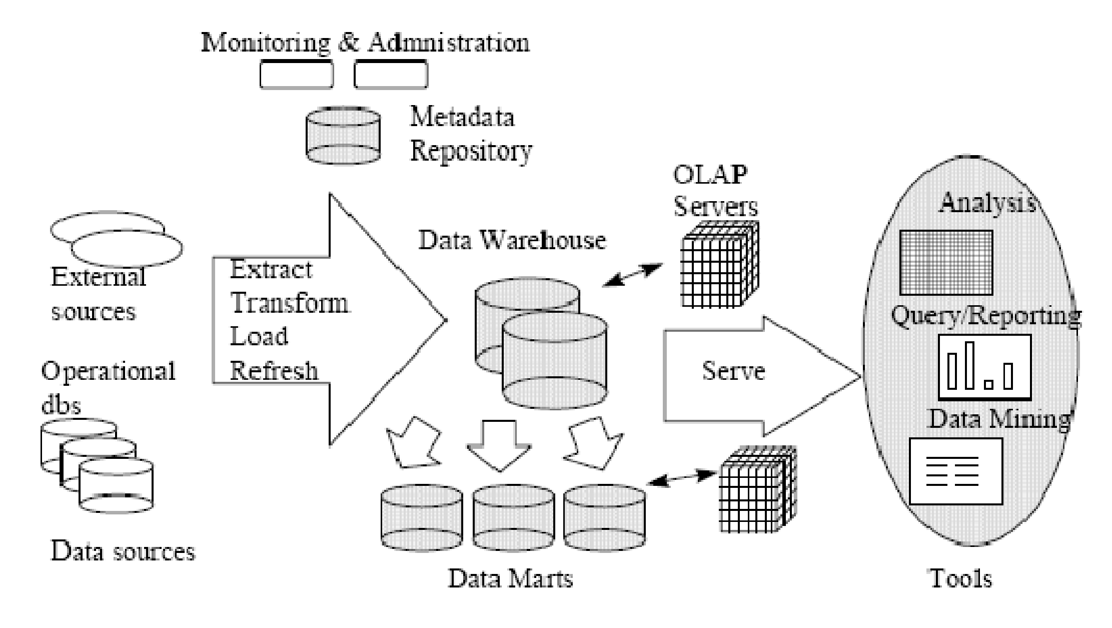


Abbildung 2-2: Data Warehouse Architektur [Chaudhury & Dayal 1997 S. 2]

Um die Daten in ein DW zu laden sind drei Arbeitsschritte notwendig. Zum einen müssen die Daten aus verschiedenen Datenbanken oder anderen externen Datenquellen entnommen werden (*Extract*). Daraufhin werden diese Daten bereinigt und in die richtige Form für die Speicherung in ein DW gebracht (*Transform*). Schlussendlich müssen die bereinigten und transformierten Daten noch in ein DW gespeichert bzw. geladen werden (*Load*). Die Vereinigung der drei genannten Arbeitsschritte wird in der Fachliteratur meist als ETL-Prozess bezeichnet [Chaudhury & Dayal 1997 S. 2ff].

Zuzüglich zum zentralen DW werden oftmals auch Data Marts bereitgestellt um Information besser auf spezielle Benutzerbedürfnisse abstimmen zu können.

Definition 2-4: *Data Marts* sind kleinere multidimensional modellierte und auswertungsorientierte Einheiten des DW und werden in der Regel für die Anwendung in einzelnen Geschäftszweigen bereitgestellt, um eine übersichtlichere Gestaltung und schnellere Abfragen zu ermöglichen [Heinrich et al. 2004 S. 164].

Die Daten eines DWs werden in einem oder mehreren DW-Servern gespeichert, welche verschiedene Sichten für die Anwendung in Front-End Werkzeugen auf die Daten bereitstellen können. Die Implementierung kann grundsätzlich auf zwei verschiedene Weisen erfolgen:

- Durch relationale OLAP Server (ROLAP). Dies bedeutet, dass durch die Speicherung der Daten in relationale Datenbanken SQL als Abfragesprache unterstützt wird. Die Server unterstützen oftmals auch Erweiterungen zu SQL.
- Durch multidimensionale OLAP Server (MOLAP) werden die Daten direkt in multidimensionalen Konstrukten (Arrays) in der Datenbank gespeichert. Die OLAP Funktionen können somit direkt auf die multidimensionalen Konstrukte zugreifen. [Bauer & Günzel 2004 S. 121f]

Diese Arbeit bezieht sich in weiterer Folge ausschließlich auf das ROLAP Modell.

Zusätzlich werden im Metadaten-Repository die Metadaten des DWs verwaltet [Chaudhury & Dayal 1997 S. 2ff]. Das Metadaten-Repository beinhaltet Information welche Aufbau und Wartung des Administrationsprozesses unterstützt und ohne welche die Verwendung von Query Werkzeugen unmöglich wäre. Die Metadaten gliedern sich in fachliche und technische. Erstere beschreiben, wie das DW-Schema zu interpretieren ist und zeigen folglich wie effektive Analyseanwendungen durchgeführt werden müssen. Technische Metadaten sind für Administratoren und Entwickler interessant. Sie beschreiben u.a. die verwendeten logischen und physischen Datenschemata [Bauer & Günzel 2004 S. 68f].

Definition 2-5: *Metadaten* sind Daten über Daten. Im Zusammenhang mit einem DW beschreiben sie alles was ein DW definiert (Tabelle, Spalten, Abfragen, Ausgabe, Geschäftsregeln und Algorithmen) [Gardner 1998 S. 59].

Definition 2-6: Ein *Repository* bezeichnet im weitesten Sinne alles was zur Aufbewahrung dient. DW betreffend wird es in der Regel zur Speicherung von Daten und Katalogen von Metadaten verwendet [Heinrich 2004 S. 563].

2.1.4 Multidimensionales Datenmodell

Das multidimensionale Datenmodell bildet die Grundlage für das DW, als auch für die darauf aufbauenden Data Marts und die zugreifenden OLAP Abfragen. In der Literatur finden sich eine Menge Vorschläge wie die Modellierung durchgeführt werden sollte. Doch es konnte

sich bis zum jetzigen Zeitpunkt kein Ansatz dauerhaft in der Praxis durchsetzen [Luján-Mora et al. 2002 S. 201ff].

Grundlage des multidimensionalen Datenmodells bildet der Würfel. Die Kanten des Würfels bilden die Dimensionen und der Inhalt des Würfels formt sich aus den, den Fakten entnommenen, Kenngrößen (engl. Measures). Die Länge einer Kante ergibt sich aus der Anzahl der Instanzen der verwendeten Dimensionen. Der in Abbildung 2-3 angezeigte Würfel enthält drei verschiedene Dimensionen und wird dementsprechend als Quader dargestellt. Sollte ein Würfel nur aus zwei Dimensionen bestehen, würde in der Abbildung eine einfache Tabelle dargestellt werden. Ein Würfel kann beliebig viele Dimensionen beinhalten [Bauer & Günzel 2004 S. 105].

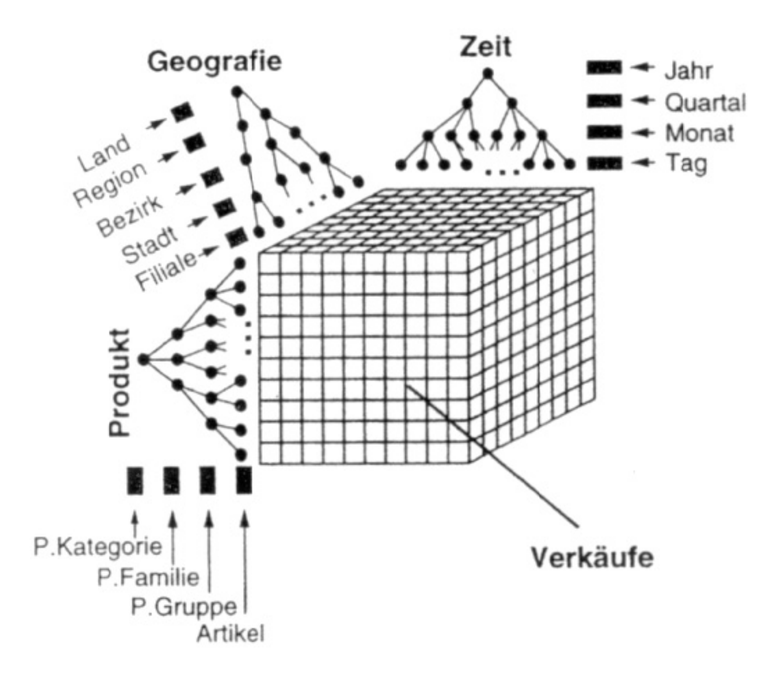


Abbildung 2-3: Modell eines Würfels [Bauer & Günzel 2004 S. 103]

Definition 2-7: Ein *Würfel* bildet die Grundlage für ein multidimensionales Modell. Die Kanten des Würfels bilden die Dimensionen. In den Zellen des Würfels wird zumindest eine Kenngröße abgelegt [Bauer & Günzel 2004 S. 105].

Definition 2-8: *„Im OLAP-Themenbereich wird unter einer Dimension eines Raums eine ausgewählte Entität verstanden, mit der eine Auswertungssicht eines Anwendungsbereiches definiert wird und die der eindeutigen orthogonalen Strukturierung dient".* [Bauer & Günzel 2004 S. 102]

Definition 2-9: Eine *Aggregationsstufe* (auch Aggregationsebene, Hierarchiestufe oder Klassifikationsstufe) ist die Schicht welche die Granularität eines Elements der Dimension definiert. Eine Dimension unterteilt sich in verschiedene Elemente, welche als Baum verknüpft die Klassifikationshierarchie definieren. Die Baumhierarchie beschreibt eine Menge von 1:n Beziehungen, wobei jedes Dimensionselement bei niedriger Granularität in mehrere Dimensionslevel unterteilt werden kann. Die höheren Aggregationsstufen beinhalten die aggregierten Werte der unteren Stufen. Die Aggregationsstufe beschreibt den Verdichtungsgrad. Der untersten Aggregationsstufe werden die Werte für die Kenngrößen hinterlegt [Bauer & Günzel 2004 S. 103].

Eine Dimension wird in der Abbildung als ein Graph in Form eines Baumes dargestellt. Die einzelnen Knoten des Baumes stellen die verschiedenen Dimensionselemente dar. Innerhalb einer Dimension muss eine Hierarchie vorhanden sein. So gliedert sich die Dimension Geografie in die Ebenen Filiale, Stadt, Bezirk, Region und Land, wobei Filiale die detaillierteste Aggregationsebene ist.

Die Hierarchie besteht aus einer Menge von Roll-up Funktionen. Durch eine Roll-up Funktion werden zwei Dimensionselemente aufgrund der Granularität zusammengeführt. So ist „Stadt" z.B. ein Teilelement von „Bezirk", was einer 1:n Beziehung entspricht. In der Hierarchie jeder Dimension muss weiters ein Basis- bzw. Grundelement vorhanden sein. Dies ist ein Dimensionselement, welches nicht mehr aufgespalten werden kann. Im Falle der Dimension „Produkt" würden sich diese Elemente in der Aggregationsstufe „Artikel" befinden. Aus der Kombination von Dimensionselementen kann in der Folge auf eine bestimmte Menge an Kenngrößen zugegriffen werden [Cabibbo & Torlone 2004 S. 2].

Definition 2-10: Eine *Roll-up Funktion* greift anhand der Hierarchie einer Dimension auf das Dimensionselement einer höheren Stufe zu. Je höher die Stufe der Hierarchie desto weiter werden die Daten zusammengefasst [Bauer & Günzel 2004 S. 102].

Definition 2-11: Eine *Kenngröße* bzw. eine bestimmte Menge an Kenngrößen ist das eigentliche Resultat, das der Benutzer aus dem Würfel auslesen will. Es sind also die Elemente die durch die verschiedenen Dimensionen beschrieben werden. Die Werte in Kenngrößen sind üblicherweise numerisch.

Durch das multidimensionale Datenmodell entstehen für die Entwicklung von DWs einige Vorteile. Zum einen liegt dieses Modell nahe am Gedankenmodell eines Datenanalysten und zum anderen ermöglicht es Performancesteigerungen, da der Entwickler, aufgrund des relativ einfachen Modells die Möglichkeit hat, häufig gestellte Abfragen vorauszusagen [Abelló et al. 2006 S. 1].

2.1.5 Erstellung eines Data Warehouses

[Abelló et al. 2006 S. 1] unterteilt die Entwicklung bzw. das Design von DWs in vier grundlegende Phasen.

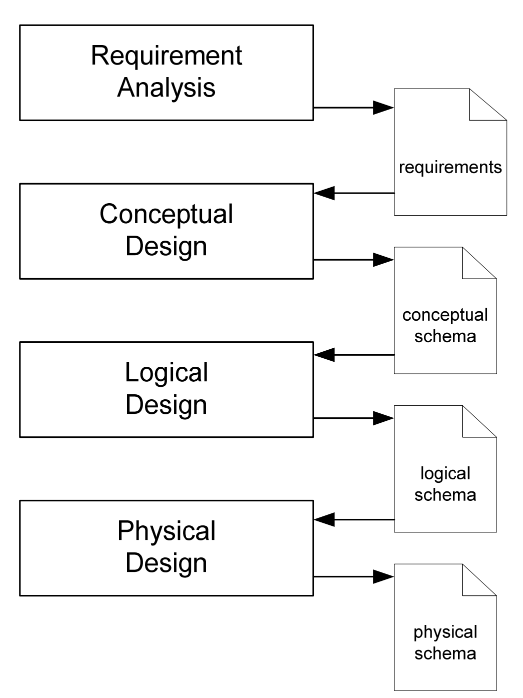


Abbildung 2-4: Grundphasen Design des Data Warehouse [Abelló et al. 2006 S. 1]

Anforderungsanalyse (engl. Requirement Analysis)

"Data warehouses are built to answer specific business problems, not to showcase the wonders of technology". [Gardner 1998 S. 60]

Ein DW soll Antworten auf die betriebliche Fragestellungen liefern und nicht nur die technischen Möglichkeiten ausreizen. Daraus folgend werden die Kundenbedürfnisse oft zu wenig in den Entwicklungsprozess miteinbezogen. Dies ergibt sich unter anderem daraus, dass Entwickler die Probleme der Benutzer nicht verstehen bzw. Benutzer nicht in der Lage sind deren Anforderungen entsprechend zu formulieren. Techniker vergessen oft im Antlitz der technischen Möglichkeiten sich auf die grundlegenden Bedürfnisse zu beschränken [Gardner 1998 S. 60].

[Hüsemann et al. 2000 S. 5] empfiehlt zur Festlegung der Kundenbedürfnisse einen tabellarischen d.h. einen zumindest semi-formalen Ansatz, bei welchem der Designer und Experte für die Geschäftsprozesse relevante Attribute aus der betrieblichen Datenbank definiert und diese in Fakten und Dimensionen bzw. Dimensionslevel unterteilt.

Konzeptueller Entwurf (engl. Conceptual Design)

Zur Bildung des konzeptuellen Entwurfs muss aus der meist informellen Spezifikation der Benutzerbedürfnisse eine formelle Definition erstellt werden. Ein wichtiger Punkt hierbei ist, die Unabhängigkeit des erstellten Schemas von jeglicher Implementierung. Die Schwierigkeit liegt auf der korrekten Erstellung bzw. Zuordnung der Daten zu Dimensionen und Fakten [Abelló et al. 2006 S. 1].

Die Erstellung eines konzeptuellen Schemas teilt sich in drei sequentielle Schritte:

1. Definition der funktionalen Abhängigkeiten zwischen Kenngrößen, Dimensionen und Aggregationsstufen der Dimension.
2. Erstellung der Hierarchie innerhalb der Dimensionen aufgrund der vorhandenen Aggregationsstufen.
3. Definieren von Beschränkungen auf die Aggregation von Kenngrößen, um sinnlose Ergebnisse zu vermeiden. [Hüsemann et al. 2000 S. 6, Golfarelli et al. 1998 S. 2ff]

Logischer Entwurf (engl. Logical Design)

Im logischen Entwurf wird das konzeptuelle Schema in ein logisches Schema zu transformiert und die Entscheidung für ein bestimmtes Implementierungsmodell getroffen. Aus diesem Grund muss zwischen einem relationalen und einem multidimensionalen Implementierungsmodell unterschieden werden. In dieser Arbeit wird, wie bereits erwähnt, nur auf das relatio-

nale eingegangen. Zusätzlich gilt es eine Entscheidung zwischen Snowflake- und Star-Schema zu treffen [Golfarelli & Rizzi 1998 S. 8].

Ein Star-Schema bildet sich aus einer Faktentabelle und einer weiteren Tabelle für jede Dimension. Die Vorteile eines Starschemas gegenüber eines Snowflake-Schemas liegen in der Performance, wofür eine redundante Speicherung von Daten notwendig ist [Chaudhury & Dayal 1997 S. 5]. Die höhere Performance ergibt sich aus der geringeren Anzahl an Tabellen und der damit verbundenen einfacheren Abfragen. Wie man in Abbildung 2-5 erkennen kann, wird beim Star-Schema die Anzahl der verwendeten Tabellen minimiert. Im Vergleich zum Snowflake-Schema ist das Star-Schema in der Regel leichter überschaubar und dadurch übersichtlicher.

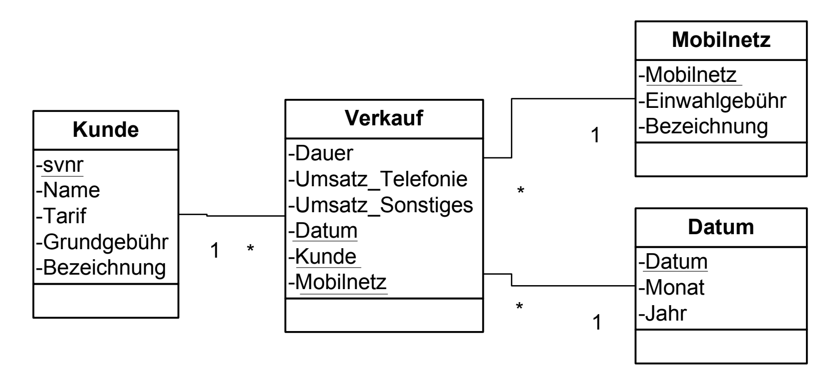


Abbildung 2-5: Star-Schema

Im Zentrum eines Snowflake-Schemas steht wiederum die Faktentabelle. Im Unterschied zum Star-Schema sind die Dimensionen jedoch in mehrere Klassifikationsstufen unterteilt. Dies hat u.a. den Vorteil das UML oder ME/R Modelle fast direkt in eine relationale Datenbank übersetzbar sind. Jede Klassifikationsstufe entspricht einer Tabelle, wobei jede Tabelle einen Fremdschlüssel auf eine andere benachbarte Tabelle mit einer 1:n Beziehung enthält. Durch die Normalisierung der Dimension ergibt sich eine einfachere Wartbarkeit, welche jedoch wiederum Probleme mit der Performance herbeiführen kann. Die Performanceprobleme ergeben sich aus der komplexeren Struktur und damit verbundenen aufwendigeren Abfragen, da das Snowflake-Schema eine höhere Anzahl an Joins zwischen den einzelnen Tabellen voraussetzt [Bauer & Günzel 2004 S. 203].

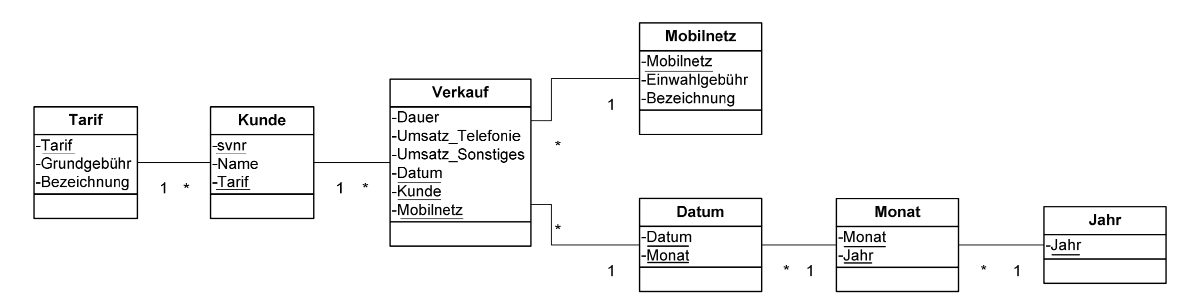


Abbildung 2-6: Snowflake-Schema

In dieser Arbeit wird in weiterer Folge ausschließlich auf das Star-Schema eingegangen.

Physischer Entwurf (engl. Physical Design)

Im Mittelpunkt der Phase des physischen Entwurfs (engl. Physical Design) steht die physische Implementierung des logischen Schemas. Dabei müssen die individuellen Einstellungen des Zielsystems inklusive jeglicher Optimierungstechniken, wie z.B. Indexierung und Partitionierung, berücksichtigt werden. Auch OLAP spezifische Optimierung wie z.B. Denormalisierung kann zum Einsatz kommen [Hüsemann et al. 2000 S. 6].

2.2 Föderierte Data Warehouses

Der Bedarf nach der Verbindung von DWs ergibt sich aus der Forderung, Information aus mehreren heterogenen Data Marts von einer zentralen Stelle auszulesen. Dies kann einerseits durch den Wunsch nach abteilungsübergreifender Information in einem Betrieb erfolgen, andererseits durch Unternehmenszusammenschlüsse oder Akquisitionen. Um auf die Daten unterschiedlicher Data Marts zugreifen zu können, muss ein übergreifendes Schema über alle Data Marts erstellt werden. Im Zuge der Erstellung des übergreifenden Schemas müssen die Heterogenitäten zwischen den verschiedenen Data Marts, wie z.B. Währungsunterschiede, überbrückt werden [Berger & Schrefl 2006].

Eine anschauliche Illustration für diese Problematik ist das in 1.3 beschriebene Rahmenbeispiel dieser Arbeit. Unternehmen MFA akquiriert das Unternehmen MFB. MFA hat den Wunsch das DW von Unternehmen MFA zu Analysezwecken mit dem DW von Unternehmen MFB zu vereinen. Aufgrund der unterschiedlichen Strukturen in den Data Marts ist es nicht möglich, einfache Abfragen über beide Datenstrukturen zu stellen. Aus diesem Grund entsteht

die Forderung Heterogenitäten zwischen den beiden Data Marts zu überwinden, um Abfragen über beide Data Marts stellen zu können.

Ein nahe liegender Ansatz die Data Marts einheitlich zu strukturieren, ist die Erstellung eines neuen DWs. Dieser Ansatz ist technisch einfach umzusetzen, jedoch verlangt er eine große Menge an Speicherplatz. Der erhöhte Speicherplatzbedarf sowie der damit verbundene erhöhte Wartungsaufwand ergeben sich aus der doppelten Speicherung der Daten. Die Daten müssen in den zu importierenden und dem globalen DW verwaltet werden.

Ein weiterer Lösungsansatz versucht die verschiedenen DWs auf Ebene der logischen Schemata durch das Anlegen eines föderierten DW Systems zu verbinden. Dieser Ansatz erfordert keine separate physische Datenbank bzw. DW. In dieser Arbeit wird in weiterer Folge ausschließlich auf diesen Ansatz zur logischen Schema Integration eingegangen.

Anfangs werden die Grundlagen verteilter Datenbanksysteme besprochen, um in weiter Folge auf die Umsetzung eines föderierten DWs zu schließen. In Anschluss wird auf den in dieser Arbeit verwendeten Ansatz des föderierten DW Systems eingegangen. Schlussendlich wird das Überbrücken von Heterogenitäten innerhalb eines föderierten DWs erörtert.

2.2.1 Verteilte Datenbanksysteme

Durch die zunehmende weltweite Vernetzung, sowie der immer moderner werdenden Kommunikationsnetze steigt der Einfluss von verteilten Datenbanken auf den Wirtschaftssektor kontinuierlich an.

Definition 2-12: Eine *verteilte Datenbank* definiert sich aus der Ansammlung verschiedener Informationseinheiten, welche auf mehreren Rechnern verteilt über ein Kommunikationsnetz miteinander verbunden sind. [Kemper & Eickler 2006 S. 449]

Der grundlegende Ausgangspunkt eines verteilten Datenbanksystems ist das globale Schema, welches durch das zwischengesetzte Fragmentierungsschema als auch das Allokationsschema auf das lokale Schema verteilt wird. In Abbildung 2-7 wird die Referenzarchitektur eines verteilten Systems grafisch dargestellt. In dieser Abbildung sind die Ortsabhängigkeiten der verschiedenen Elemente hervorgehoben. Das globale Schema entspricht aufgrund dieser Archi-

tektur im weiteren Sinne dem konsolidierten relationalen Implementierungsschema des zentralisierten Datenbankentwurfs.

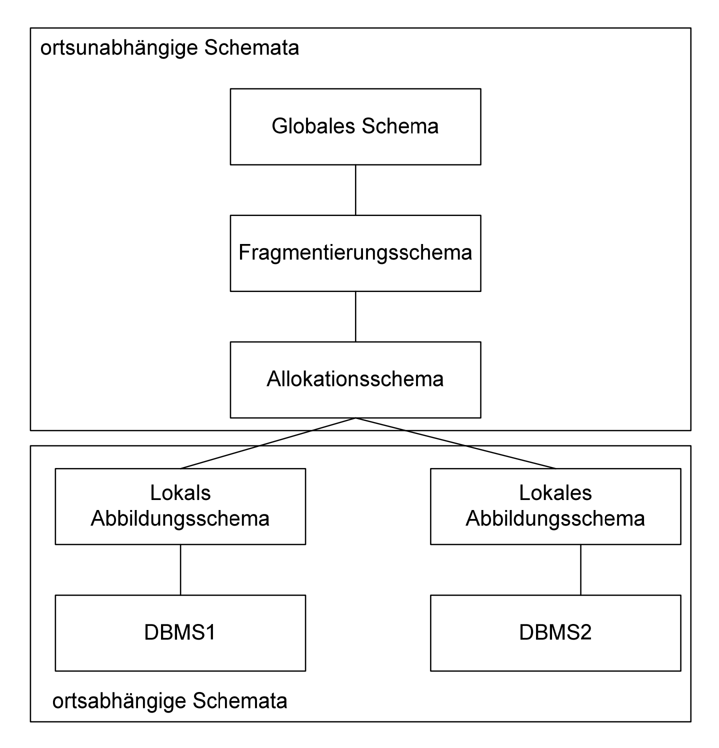


Abbildung 2-7: Referenzarchitektur für verteilte DBS (angelehnt an [Kemper & Eickler 2006 S. 451])

Durch die Fragmentierung werden logische zusammengehörige Informationsmengen auf Grundlage des Zugriffsverhaltens in verschiedene, disjunkte Untereinheiten zerteilt. Die Umsetzung dieses Schemas erfordert ein umfassendes Wissen über die zukünftigen auf der Datenbank ausgeführten Anwendungen.

Aufbauend auf der Fragmentierung wird die Allokation durchgeführt. Es werden die verschiedenen bei der Fragmentierung erzeugten Fragmente den einzelnen Netzwerkknoten des Datenbanksystems zugeordnet. Es wird zwischen einer redundanzfreien- und einer Allokation mit Replikation unterschieden. Der Unterschied liegt in der Anordnung von den Fragmenten. Bei der redundanzfreien Allokation wird im Gegensatz zur Allokation mit Replikation kein Fragment doppelt gesichert. [Kemper & Eickler 2006 S. 451]

Durch den Einsatz des globalen Schemas sind die Anwendungsprogramme von der physischen Speicherung der Daten entkoppelt. Dies bedeutet, dass eine hohe Datenunabhängigkeit gegeben ist und die Anwendungen nicht direkt auf die einzelnen lokalen Datenbanken zugrei-

fen müssen. Durch diesen Umstand sind Maßnahmen zur Effizienzsteigerung in der Datenbank bzw. in den Datenbanksystemen möglich, ohne Änderungen an den Anwendungen vorzunehmen. [Dadam 1996 S. 83]

2.2.2 Föderierte Datenbanksysteme

Ein föderiertes Datenbanksystem besteht aus einer Menge von interagierenden aber autonomen Datenbanksystemen. Diese können entweder aus einer einfachen Datenbank, einem verteilen Datenbanksystem oder einem anderen föderierten Datenbanksystem bestehen. Um eine ordnungsgemäße und koordinierte Bearbeitung der föderierten Datenbanksysteme zu gewährleisten ist ein föderiertes Datenbankmanagementsystem unumgänglich. Zur Hauptaufgabe des Datenbankmanagementsystem zählt es, verschiedene unabhängige Systeme zu verbinden [Sheth & Larson 1990 S. 183].

Definition 2-13: *Ein **föderiertes Datenbanksystem** ist ein System das aus mehreren ursprünglich voneinander unabhängigen Datenbanksystemen entwickelt wird, indem eine Datenbankfunktionalität zur Verfügung gestellt wird, die einen integrierten Datenbestand schafft.* [Heinrich 2004 S. 261]

Auf mehreren Datenbanksystemen basierende Systeme, von welchen die föderierten Datenbanksysteme ein Teil sind, können nach drei verschiedenen Kriterien charakterisiert werden.

- **Distribution**: Daten können auf unterschiedliche Datenbanken aufgeteilt werden, welche wiederum auf verschiedene Computer verteilt sein könnten.
- **Heterogenität**: Die Heterogenitäten lassen sich in die Bereiche technologische Unterschiede und Unterschiede in der Struktur der Daten unterteilen. Zu den technologischen Unterschieden zählen u.a. Unterschiede in der Hardware, Betriebssystem und Kommunikationstechnologien. Der häufigste Problemgrund ist jedoch die unterschiedliche Repräsentation von Entitäten in den Strukturen des Schemas.
- **Autonomie**: Da in föderierten Datenbanken verschiedene Datenbanksysteme von verschiedenen Anbietern inkludiert werden können, ist eine hohe Autonomie der einzelnen Systeme gegeben. Dies bedeutet, dass die angebundenen Systeme von unterschiedlichen Betreibern oder Administratoren erstellt bzw. gewartet werden. Oftmals werden die Zugriffsrechte für föderierte Datenbanksysteme von den Betreibern der

jeweiligen Systeme stark eingegrenzt. Aufgrund der Autonomie entstehen oftmals die bereits erwähnten Heterogenitäten im Aufbau der verschiedenen Systeme.

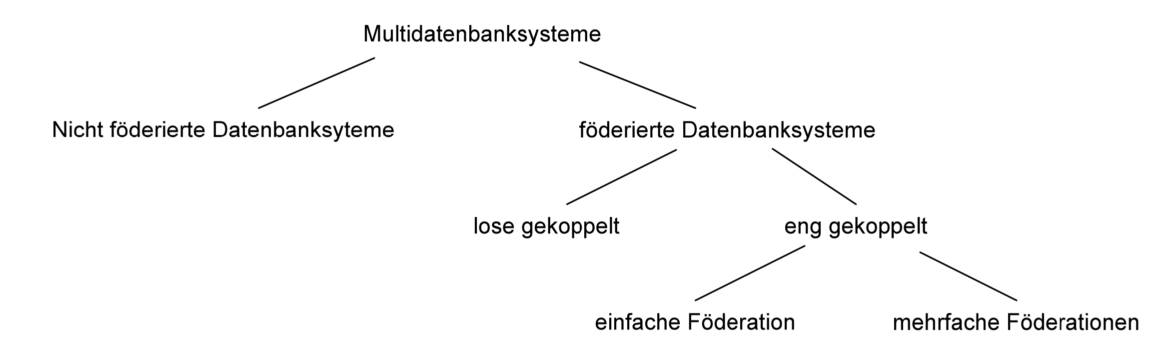


Abbildung 2-8:Einordnung von föderierten Datenbanken (angelehnt an [Sheth & Larson 1990 S. 190])

Abbildung 2-8 stellt die Einordnung bzw. die Unterscheidung von verschiedenen Multidatenbankensystemen und im Speziellen von föderierten Datenbanken dar. Zu den Multidatenbanken zählen alle Systeme welche mehrere Datenbanksysteme miteinander verbinden. Grundsätzlich kann ein Multidatenbanksystem homogen (alle Datenbanken sind gleich aufgebaut) oder heterogen sein. Der Unterschied zwischen föderierten und nicht föderierten Datenbanksystemen ergibt sich aus der Autonomie. Nicht föderiere Datenbanksysteme haben nur eine Managementinstanz, welche alle Datenbanksysteme bearbeitet. Im Gegensatz dazu sind die Datenbanksysteme eines föderierten Datenbanksystems autonom.

Föderierte Datenbanksysteme unterscheiden sich in lose gekoppelte (engl. loosely coupled) und in eng gekoppelte (engl. tightly coupled) Systeme. Die Unterscheidung ergibt sich aus den Verantwortungen für das System. So trägt bei lose gekoppelten Systemen der Benutzer die Verantwortung, festzustellen wie welche Datenbanksysteme an der Föderation teilnehmen und wie die Heterogenitäten zwischen diesen überwunden werden. Dies bedeutet, dass Administrator nicht in den Prozess eingreift und der Benutzer selber die Verbindung der unterschiedlichen Datenbanken festlegt. Ein System wird als eng gekoppelt angesehen, wenn der Administrator die Verantwortung für die Erstellung und die Wartung übernimmt.

Eng gekoppelte Systeme können in weiterer Folge wiederum in einfache Föderationen oder mehrfache Föderationen unterteilt werden. Die Differenzierung zwischen diesen Merkmalen besteht in der Anzahl der auf der globalen Schicht verwendeten föderierten Datenbanksyste-

me. Durch die Verwendung mehrerer globaler Datenbanksysteme kann die Komplexität besser auf die unterschiedlichen Datenbanksysteme verteilt werden. Dies erfordert jedoch aufgrund der höheren Anzahl an Elementen einen größeren Wartungsaufwand. [Sheth & Larson 1990 S. 190ff]

2.2.3 Föderierte Data-Warehouses

Ein föderiertes DW wendet die Idee von föderierten Datenbanken auf DWs an. Zentrales Element eines föderierten DW ist eine übergeordnete Schicht, welche als Grundlage für OLAP Anfragen gilt. Durch diese Schicht wird die Heterogenität zwischen den verschiedenen autonomen Data Marts ausgeglichen. Dem Benutzer wird ein „virtueller" Würfel bereitgestellt, auf welchen er die Abfragen stellen kann. Durch das Benützen eines „virtuellen" Würfels bzw. durch das Definieren eines globalen Schemas wird eine enge Koppelung der verschiedenen Data Marts erreicht [Berger & Schrefl 2006].

Definition 2-14: Ein *föderiertes Data Warehouse* kennzeichnet sich durch die Verbindung von verschiedenen Data Marts auf Schemaebene. Die vom Administrator beseitigten Heterogenitäten werden dem Benutzer durch eine übergeordnete Schicht vorenthalten.

Im Vergleich zu föderierten Datenbanken kann die Integration von Data Marts auf einem systematischeren Weg bewältigt werden. So sind die Daten in einem Data Mart weitaus einheitlicher gegliedert, als in den meisten Datenbanken. Zusätzlich ist die Qualität an Daten höher, da durch den ETL-Prozess bereits eine gewisse Reinigung der Daten erfolgt ist. Aufgrund dieser Gegebenheiten konzentriert sich die Integration von autonomen Data Marts auf die unabhängig voneinander entwickelten Dimensionen und Fakten [Cabibbo & Torlone 2004].

In dieser Arbeit wird der Ansatz eines föderierten DW Systems von [Berger & Schrefl 2006] herangezogen. Die Architektur eines föderierten DWs gliedert sich in vier Ebenen (siehe Abbildung 2-9).

Auf der untersten Ebene befinden sich die zu integrierenden Data Marts (**Datenquellen**). Diese werden in weiterer Folge von dem lokalen Datenmodell (*Lokales DW-Schema*) in ein vereinheitlichtes Datenmodell (*Komponenten-Schema*) umgewandelt. Aufgrund des einheitlichen

Datenmodells ist es möglich Vergleiche zwischen den unterschiedlichen Data Marts zu ziehen.

In der **Transformationsebene** werden die Komponenten-Schemata in einheitliche auf die im globalen Schemata benötigten Elemente der lokalen Schemata reduziert. Die daraus entstandenen *Export-Schemata* werden daraufhin in *Import-Schemata* transformiert. Hierzu werden Mappings anhand einer Fakt- bzw. Dimensions-Algebra erstellt (siehe [Berger & Schrefl 2008]). Ein Mapping stellt eine Zuordnungsvorschrift zwischen einem Element im globalen und im importierten Schema dar.

Die **Föderationsebene** beinhaltet die zentralen Elemente eines föderierten DWs, wobei die verschiedenen Fakt- und Dimensionsschemata aus den importierten Data Marts richtig interpretiert werden müssen. Die Dimensionsschemata werden zu globalen Dimensionen zusammengefügt und im Dimension-Repository gespeichert.

Mithilfe der Mappings sollen die Unterschiede bzw. Heterogenitäten zwischen den Schemas überbrückt werden. Bei der Erstellung des globalen Schemas ist speziell auf die Bedürfnisse der Benutzer bzw. der darauf zugreifenden Applikation einzugehen. Ist das globale Schema zu allgemein gefasst, kann möglicherweise zu wenig Information daraus gezogen werden. Das globale DW-Schema stellt, wie bereits erwähnt, die Schnittstelle zu anderen Applikationen **(Applikationsebene)** dar. Im theoretischen Ansatz sollte das globale Schema keine Einschränkungen bezüglich der Applikationen vornehmen und eine möglichst allgemeine Schnittstelle anbieten. [Berger & Schrefl 2008]

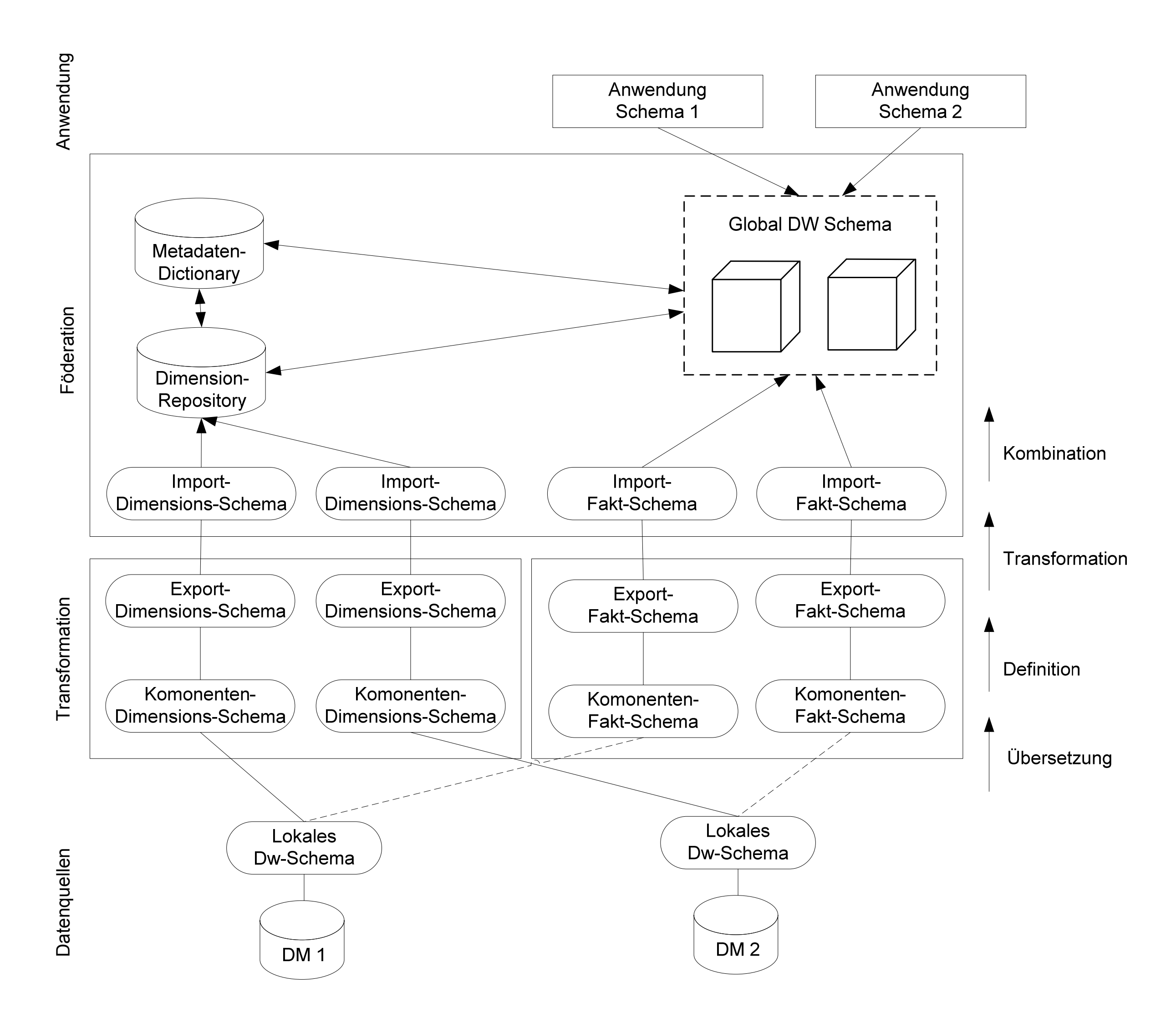


Abbildung 2-9: Architektur eines föderierten Data Warehouses (basierend auf [Berger & Schrefl 2006])

2.2.4 Mapping zwischen Dimensionen

Ein spezielles Augenmerk beim Abbilden von den Importschemata auf das globale Schema ist das Mapping von heterogenen Dimensionen. Ein anschauliches Beispiel für Schwierigkeiten beim Abgleich von Dimensionen wird in Abbildung 2-10 dargestellt.

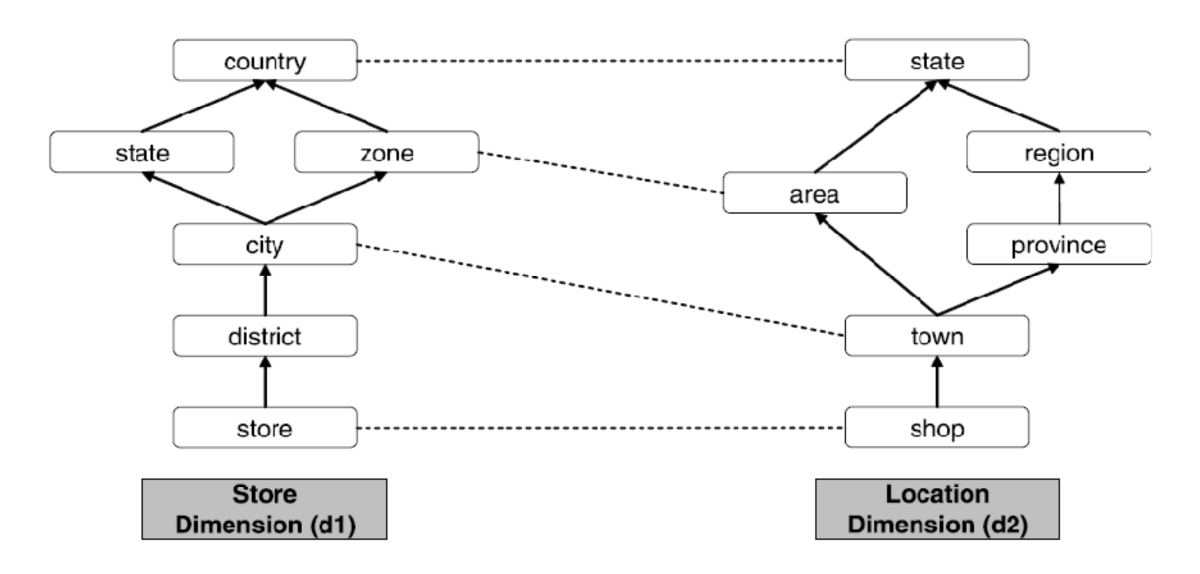


Abbildung 2-10: Mapping zwischen zwei Dimensionen [Torlone 2008 S. 77]

Voraussetzung für das Abbilden einer Dimension auf die andere ist, dass nur Aggregationsstufen herangezogen werden, welche bei beiden Dimensionen vorhanden sind. Zu den Schwierigkeiten im oberen Beispiel zählt z.B., dass im Store (d1) die Aggregationsebene *district* vorhanden ist, welche in Location (d2) nicht vorhanden ist. Dieses Problem könnte durch das Ausblenden dieser Aggregationsebene gelöst werden. Das Ausblenden ist möglich, da sich aus den Abhängigkeiten *store → district* und *district → city* transitiv *store → city* ableiten lässt. Die Zusatzinformation Aggregationsebene *district* geht demnach verloren [Torlone 2008 S. 76f].

Wichtig ist, dass die Aggregationsebenen eine inhaltliche Ähnlichkeit aufweisen bzw. inhaltliche Differenzen automatisiert beseitigt werden können. Ein Beispiel für eine solche Problematik wäre der Versuch ein amerikanisches DW mit einem österreichischen DW zu verbinden. Hierbei treten mit hoher Wahrscheinlichkeit Währungsdifferenzen auf. Aufgrund der Differenzen können die Dimensionen nicht direkt verglichen werden. Eine Möglichkeit diese Problematik zu entschärfen ist die Verwendung von in der Datenbank vorhandenen Prozeduren (*Stored Procedures*), durch welche eine Umrechnung der Währungen erfolgen könnte. Weitere Unterschiede ergeben sich aus den Ortsangaben. Die österreichische Darstellung (*Stadt → Bezirk → Bundesland*) unterscheidet sich von der amerikanischen Darstellungsweise (*city → state → division → region*). Um beide Darstellungsweisen vergleichen zu können, müssen die Gemeinsamkeiten herausgefiltert werden. So entspricht das amerikanische *state* in etwa dem österreichischen *Bundesland*. Für einen österreichischen *Bezirk* gibt es kein ver-

gleichbares amerikanisches Gegenstück, demnach muss dieser ausgeblendet werden. Analog dazu verhalten sich *division* und *region* der amerikanischen Darstellungsweise.

2.2.5 SQL-MDi

SQL-MDi steht für „SQL for multi-dimensional integration" und dient als Grundlage für die Ausführung einer OLAP Abfrage, welche z.B. in SQL ausgeführt werden kann. Aufgabe von SQL-MDi ist es demnach Heterogenitäten zwischen verschiedenen Data Marts zu beseitigen und so einen virtuellen Würfel bereitzustellen auf welchen in der Folge die Abfragen gestellt werden können [Brunneder 2008 S. 113].

Das Grundgerüst der Sprache besteht aus folgendem Konstrukt (vgl. [Berger & Schrefl 2004 S. 125]):

```
DEFINE [GLOBAL] CUBE
MERGE DIMENSIONS
MERGE CUBES
```

Mittels des „DEFINE [GLOBAL] CUBE" Blocks werden importierte Schemata an das globale Schema angepasst. Für jedes importierte Schema und für das globale Schema muss eine CUBE-Anweisung vorhanden sein. Innerhalb dieses Blocks können folgende Aktivitäten durchgeführt werden:

- Ausblenden von Dimensionen und Kenngrößen,
- Ausblenden von Ebenen aus einer Hierarchie,
- Umbenennen von Dimensionen, Kenngrößen, Fakten, dimensionalen Attributen und Klassifikationsstufen,
- Konvertieren von Kenngrößen, Fakten und dimensionalen Attributen,
- Roll-up eines dimensionalen Attributs und
- Transformation von Fakttabellen zur Integration von heterogenen Fakttabellen.

Durch den „MERGE DIMENSIONS" Block können bestehende Schema- und Instanzkonflikte mittels einiger Sub-Anweisungen bereinigt werden. Für jede in den globalen Würfel integrierte Dimension muss eine MERGE DIMENSIONS Anweisung vorhanden sein. Folgende Konfliktlösungen können durchgeführt werden:

- Umbenennen eines nicht dimensionalen Attributs,
- Berichtigung von heterogenen Roll-up Funktionen,
- Auflösung von Präferenzen bei überlappenden Dimensionsattributen verschiedener Würfel,
- Konvertierung der Werte eines nicht-dimensionalen Attributs und
- Umbenennung von Dimensionsinstanzen.

Der „MERGE CUBES“ Block wird pro Würfel einmal benötigt und ist dafür zuständig, alle Elemente zusammenzuführen und dadurch den endgültigen „virtuellen Würfel“ zu erzeugen:

- Aggregation von überlappenden Fakten,
- Überlappende Fakten mit einer Kontextdimension erweitern und
- Handhabung von disjunkten Fakten.

2.3 Zusammenfassung

In diesem Kapitel wurden die grundlegenden theoretischen Konzepte zur Umsetzung des Werkzeuges aufgearbeitet. Hierbei wurde im Speziellen eine Einführung in DWs allgemein als auch in föderierte DWs gegeben.

Im Zentrum eines DWs steht das multidimensionale Datenmodell, welches grundlegend aus Würfel, Fakt, Dimension, Kenngröße und Aggregationsstufe besteht. Weiters müssen für die Erstellung eines DW die Phasen Anforderungsanalyse, konzeptueller Entwurf, logischer Entwurf und physischer Entwurf durchlaufen werden.

Zum Verständnis föderierter DWs sind grundlegende Konzepte aus dem Bereich verteilter und föderierter Datenbanksysteme hilfreich. Zu diesem Thema wurden eine Einführung sowie eine Definition der wichtigsten Begriffe gegeben. Nach Beschreibung des föderierten Data-Warehouses an sich, wurde in weiter Folge die Problematik der Heterogenitäten in verschiedenen Data Marts erläutert und schlussendlich der praktische Lösungsansatz SQL-MDi vorgestellt.

3 Grundlagen modellgetriebene Entwicklung

Dieses Kapitel beschreibt die Grundlagen zur Erstellung einer modellgetriebenen Softwarearchitektur mit den von der OMG empfohlenen Technologien MOF, CWM und UML. Zu Beginn wird auf die OMG und die Geschichte der modellgetriebenen Architektur (*engl. Model Driven Architecture, MDA*) sowie deren Ziele eingegangen. Anschließend werden die Grundlagen zu Metamodellen und der eingesetzten Technologien MOF und UML erläutert.

3.1 Object Management Group (OMG)

Die OMG wurde im Jahr 1989 gegründet und ist mit über 800 Mitgliedern (darunter IBM, Apple, Microsoft, Sun) das weltweit größte Softwarekonsortium. Aufgabe dieses Konsortiums ist die Weiterentwicklung und das Veröffentlichen von implementierungsunabhängigen Standards in der objektorientierten Programmierung.

Zu den Kernzielen zählen u.a. Verbesserungen in der Wiederverwendbarkeit, der Portabilität sowie der Interoperabilität von objektorientierten Softwarekomponenten. Zu den bekanntesten Entwicklungen zählen die Common Object Request Broker Architecture (CORBA, siehe [OMG CORBA 2009]), die Model Driven Architekture (MDA, siehe [OMG MDA 2009]), das XML Metadata Interchange Format (XMI, siehe [OMG XMI 2009]), Common Warehouse Metamodel (CWM, siehe [OMG CWM 2009]) sowie die Unified Modelling Language (UML, siehe [OMG UML 2009]). In dieser Arbeit wird in weiterer Folge explizit auf UML, CWM und MDA eingegangen. [OMG 2000]

3.2 Geschichte und Ziele der Model Driven Architecture

In der Geschichte der Softwareentwicklung ist eine Tendenz zu immer stärkerer Abstrahierung erkennen. Durch die Abstrahierung können komplexe Probleme in eine leichter verständliche Form gebracht werden und daraus folgend schneller gelöst werden. Diese Entwicklung wird in Abbildung 3-1 dargestellt. Im Laufe der Zeit wurden sukzessiv abstraktere Konzepte zur Lösung von Problemen eingeführt. Gegenwärtig rückt die Anwendung von Model-

len verstärkt in den Mittelpunkt der Entwicklung, insbesondere in der Phase der Analyse und des Entwurfs [Gruhn et al. 2006 S. 12ff].

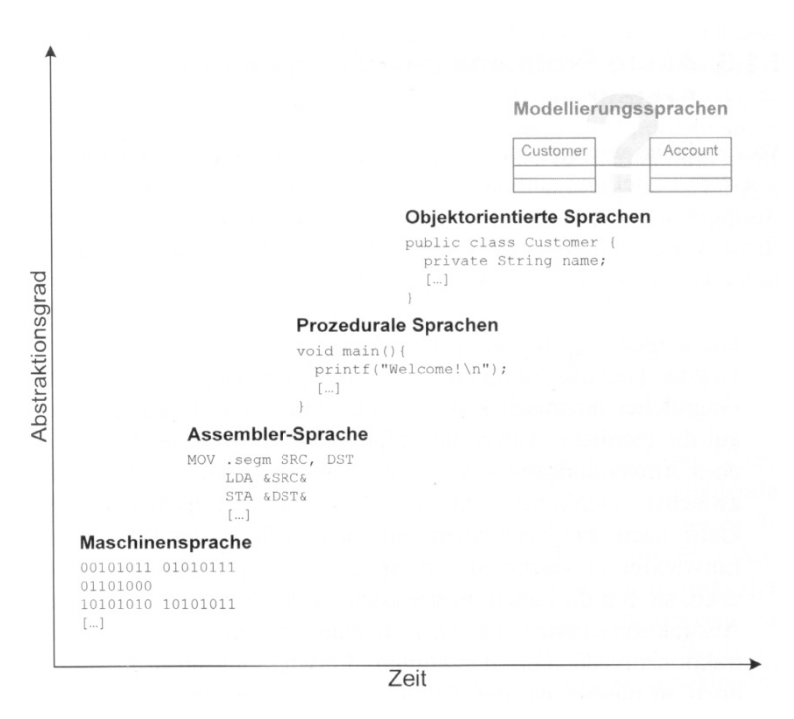


Abbildung 3-1: Steigerung des Abstraktionsgrades [Gruhn et al. 2006 S. 15]

Im Kern des Ansatzes der modellgetriebenen Architektur steht die Grundidee, dass die Spezifikation der Softwarekomponente unabhängig von der technischen Implementierung beschrieben werden sollte.

[Gruhn et al. 2006 S. 21ff] fasst fünf Ziele der MDA zusammen:

- **Konservierung der Fachlichkeit:** Durch die Verwendung von Modellen sind Systeme einfacher dargestellt und dadurch auch leichter für fachlich geschulte Experten verständlich. Bei der Umsetzung der Modelle geht dadurch die Verständlichkeit für die Experten nicht verloren.
- **Portierbarkeit**: Aufgrund der höheren Abstraktionsstufe können Systeme zu späteren Zeitpunkten mit geringem Aufwand auf neuere Technologien umgestellt werden.
- **Systemintegration und Interoperabilität**: Durch die explizite Trennung von fachlichen Konzepten und konkreter Repräsentation können Schnittstellen effizienter für die Anbindung von externen Programmen entwickelt werden.

- **Effiziente Softwareentwicklung**: In vielen Systemen besteht die Möglichkeit, Code automatisch aus Modellen zu generieren. Durch den Einsatz von Patterns und Referenzarchitekturen können gelöste Probleme in Zukunft schneller behandelt bzw. Lösungen einfach an andere Systeme adaptiert werden.
- **Domänen-Orientierung**: Das fachliche Domänenwissen wird, spezifisch abgegrenzt von der technischen Umsetzung, gespeichert. Dies bedeutet, dass wichtiges Wissen von Geschäftsprozessen im Unternehmen einfacher gespeichert und wiederverwendet werden kann. Die Kenntnis über die Geschäftsprozesse ist in der Regel von essentieller Bedeutung für jedes Softwareprojekt.

Die Bedeutung von Modellen steigt aufgrund ihrer kompakten Darstellung und der dennoch hohen Ausdrucksstärke kontinuierlich an. Die Entwicklung der MDA wird von der OMG vorangetrieben. Die OMG bemüht sich um die Einführung verschiedener neuer Vorgehensweisen und der damit verbundenen Vereinheitlichung des Ansatzes [Gruhn et al. 2006 S. 23].

3.3 Metamodellierung

Voraussetzung für die Anwendung der MDA ist die Möglichkeit ein System auf verschiedenen Abstraktionsebenen zu beschreiben. Zur Umsetzung werden Modelle herangezogen, die dieser Voraussetzung gerecht werden. Um den Aufbau von Modellen zu definieren werden Metamodelle eingesetzt. Zur Beschreibung von Metamodellen werden wiederum Meta-Metamodelle (Meta²-Modelle) verwendet [Gruhn et al. 2006 S. 84].

Definition 3-1: *Metamodelle* werden in der Fachliteratur oftmals als „Modelle von Modellen“ bezeichnet. Diese Definition ist jedoch nicht spezifisch genug. Richtig ist die Bezeichnung „Modelle von Modellierungsprachen von Modellen“. Diese Unterscheidung ergibt sich aus dem Grund, dass ein Modell immer nur einen Ausschnitt der erlaubten Notation eines Metamodells anzeigt. Demnach beschreiben Metamodelle die Sprache mit welcher ein Modell ausgedrückt werden kann. [Gruhn et al. 2006 S. 84f]

Ebene 3 Meta²-Modell
Ebene 2 Metamodell
Ebene 1 Modell
Ebene 0 Original
Abstraktion

Abbildung 3-2: Übersicht die Ebenen der Metamodellierung (basierend auf [Karagiannis & Kühn 2002])

Abbildung 3-2 stellt den Zusammenhang der verschiedenen Ebenen der Metamodelleriung dar. Ebene 0 repräsentiert das reale Objekt. Eine weitere Verfeinerung dieses Objekts ist nicht möglich. Die Beschreibung des Originals erfolgt mittels eines Modells (Ebene 1), welches wiederum durch ein Metamodell (Ebene 2) beschrieben werden kann. Ebene 3 wird als die abstrakteste Ebene definiert, diese kann durch eine rekursive Definition auf sich selbst beliebig erweitert werden [Karagiannis & Kühn 2002 S. 3].

In dieser Arbeit kommen zur Umsetzung der Modelle die von der OMG empfohlenen Technologien MOF (siehe Abschnitt 3.4) bzw. die zugrunde liegende UML (siehe Abschnitt 3.5) zum Einsatz. Folgende Abbildung veranschaulicht die Einordnung dieser Technologien in die Ebenen der Metamodellierung.

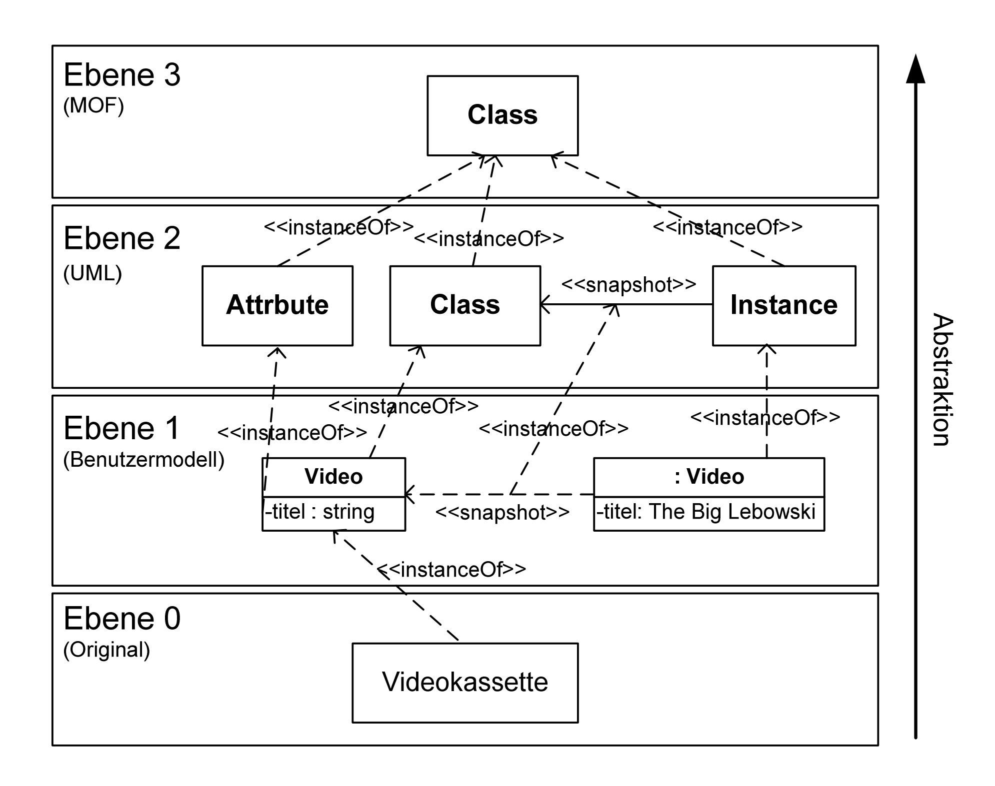


Abbildung 3-3: Beispiel für 4-Ebenen Metamodell Hierarchie (basierend auf [UML 2.0 2003])

Abbildung 3-3 zeigt die Verwendung von Metadatenkonstruktionen an einem einfachen Beispiel. Ebene 0 entspricht der Realität, welche in diesem Beispiel durch eine Videokassette dargestellt wird. Die Abstraktion dieser Videokassette erfolgt durch ein Benutzermodell, welches ein Video in Form einer Klasse sowie die Instanz des Videos darstellt. Die Regeln für diese Darstellung werden durch UML definiert (Ebene 3). Der Zusammenhang zwischen den unterschiedlichen Abstraktionsebenen wird in dieser Abbildung durch die Verbergungshierarchien *(<<instanceOf>>)* dargestellt. Die in UML verwendeten Konstrukte basieren wiederum auf den in MOF beschriebenen Regeln.

3.4 Meta Object Facility 2 (MOF)

Die Meta Object Facility 2 (MOF) spezifiziert ein gemeinsames Meta-Metamodell aller Modellierungsprachen der OMG. Demnach muss es der Ebene 3 der Abbildung 3-2 zugeordnet werden. Durch die zentrale Verwendung stellt MOF den Kernbereich der MDA von OMG dar. Durch MOF werden Sprachen definiert und erweitert. Zu den bedeutendsten MOF konformen Modellierungsprachen zählen UML (siehe Abschnitt 3.5) und das Common Warehou-

se Metamodel (siehe Abschnitt 3.6). Zusätzlich wird eine automatisierte metamodellorientiere Transformation unterstützt.

MOF 2 basiert auf dem sogenannten Core Paket der UML-Infrastructure, welches die essentiellen Bauelemente wie Klassen, Attribute, Assoziationen etc. enthält. MOF gliedert sich in die zwei Hauptbestandteile Essential MOF (EMOF) und Complete MOF (CMOF). EMOF enthält die Basisfunktionalitäten von MOF und ist aufgrund der übersichtlichen Anzahl an Metamodellen leicht verständlich. Darüberhinaus ist es für die Deklaration einfacher Metamodelle ausreichend. CMOF ist eine komplette Beschreibung des MOF Modells und ist die Grundlage für alle Metamodelle der OMG. [Uckat 2007]

3.5 Unified Modeling Language (UML)

UML ist eine von der OMG standardisierte Modellierungssprache. Die Sprache besteht aus geometrischen Formen und Diagrammtypen. Zu den Formen zählen u.a. Rechtecke, Pfeile und Ellipsen welche durch im dazugehörigen Metamodell (MOF) vordefinierte Regeln auf aussagekräftige Konstrukte zusammengeschlossen werden können. Zu den Diagrammtypen gehören beispielsweise das Klassendiagramm und das Anwendungsfalldiagramm (siehe Abbildung 3-4). UML-Modelle dienen meist als Abstraktion, welche als Grundlage für die Softwareentwicklung oder für die verständliche Aufbereitung des Codes verwendet wird. Durch die grafische Aufbereitung soll dem Entwickler die Möglichkeit gegeben werden, aus einfachen Gebilden eine Vielzahl an Information zu erschließen. In der UML-Spezifikation finden sich 13 verschiedene Diagrammtypen welchen unterschiedlichen Aufgabenbereichen entsprechen [Schäling 2005].

Grundsätzlich lassen sich die Diagrammtypen in Strukturdiagramme und Verhaltensdiagramme unterteilen. Abbildung 3-4 beschreibt die detaillierte Gliederung der verschiedenen Diagramme. Strukturdiagramme konzentrieren sich auf die statischen Betrachtungsweisen eines Systems. Im Gegensatz dazu stellen Verhaltensdiagramme die dynamischen, meist zur Laufzeit veränderten, Aspekte dar [Gruhn et al. 2006 S. 83].

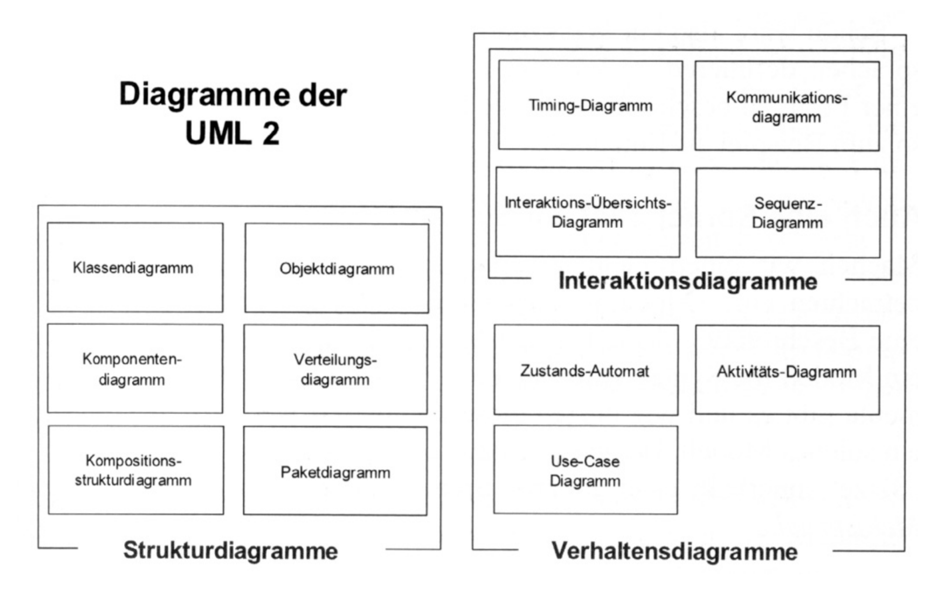


Abbildung 3-4: Diagramme Uml [Gruhn et al. 2006 S. 15]

Die für diese Arbeit relevanten Diagramme werden folgend näher beschrieben [Heinrich et al. 2004 S. 676f]:

- **Klassendiagramm**: Der am weitesten verbreiteteste Diagrammtyp von UML. Das Klassendiagramm besteht aus Klassen, Paketen, Objekten und Diagrammen, sowie deren Beziehungen zu einander. Aufgrund der Verwendung von Klassen und Objekten und der damit verbundenen hohen Ausdruckstärke eignet sich dieses Modell für den Einsatz in der Umsetzung des Metamodells.
- **Anwendungsfalldiagramm** (engl. Use-Case Diagramm): Beschreibt die Interaktion des Anwenders mit der Software. Diese Art der Dokumentation wird in einem Großteil der Fälle in Systemspezifikationen und Lastenheften verwendet.
- **Aktivitätsdiagramm**: Beschreibt das Zusammenwirken und das Zusammenspiel einzelner Aktivitäten sowie die daraus resultierenden Zustände. Der Einsatz dieser Diagramme erfolgt meist in der Systemspezifikation.

3.6 Common Warehouse Metamodell (CWM)

Das CWM ist ein von der OMG eingeführter Standard welcher Regeln für Beschreibung und Austausch von Metadaten eines DWs vorgibt. Durch den einheitlichen Standard entstehen zwar bei der Entwicklung einer Struktur für DWs höhere Kosten und Zeitaufwand, es wird

jedoch davon ausgegangen, dass aufgrund der höheren Interoperabilität zwischen verschiedenen Programmen langfristig ein besseres Ergebnis erzielt wird [Chang et al. 2002 S. 1ff].

In der Praxis werden unterschiedliche Metadaten mithilfe sogenannter „meta data bridges" verbunden. Diese sind Programme welche Metadaten eines Produktes in die Metadaten eines anderen Produktes umwandeln. Für die Erstellung ist ein fundiertes Grundwissen über beide Metadatenkonstrukte erforderlich. Für jedes weitere zu integrierende Produkt müssen daraufhin weitere Brücken erstellt werden. Durch die hohe Anzahl an verschiedenen Brücken entsteht ein System welches aufgrund der vielen unterschiedlichen Strukturen schwer zu warten und somit langfristig gesehen kostspielig ist.

Durch den Einsatz eines einheitlichen Metamodells, dem CWM, wird die Komplexität des Metadaten-Austauschs verringert. Dazu wird ein Metadaten-Repository eingesetzt, welches durch eine Datenbank dargestellt wird. In dieser Datenbank werden alle relevanten Metadaten gespeichert und den einzelnen Komponenten eines DW-Systems zur Verfügung gestellt. Daraus folgend muss die Wartung der Metadaten nur noch an einem zentralen Punkt vorgenommen werden. Um eine Kompatibilität mit dem CWM zu erreichen, erfordert es hohe Investitionen am Anfang der Entwicklung, jedoch sollte das „Return of Investment" (ROI) schon nach den ersten Integrationen und Anbindungen von weiteren Produkten erreicht werden [Chang et al. 2002 S. 1ff].

Das CWM gliedert sich in 21 unterschiedliche Pakete, welche aus Klassen, Assoziationen und Constraints bestehen und sich jeweils mit einem Teilgebiet des DWs auseinander setzen. Die Pakete verteilen sich auf fünf auf einander aufbauende Schichten [Chang et al. 2002 S. 26ff]:

- **Object Model**: Bildet die unterste Ebene des CWM und stellt die fundamentalen Konzepte, Verbindungen und Einschränkungen bereit. Das Object Model stellt eine Teilmenge der UML-Spezifikation dar. Die Klassen und Assoziationen des CWM referenzieren in der Mehrheit der Fälle direkt die Klassen und Assoziationen der UML.
- **Foundation**: Unterscheidet sich vom Object Model dadurch, dass die zur Verfügung gestellten Services speziell für das CWM entwickelt worden sind. Zu den Services gehören Funktionalitäten, welche für den Einsatz in einem DW notwendig sind wie z.B. Indexierung, Mapping oder Datentypen.

- **Resource**: Beschreibt die Struktur von Datenressourcen, welche als Ziel- oder Ausgangspunkt eines auf CWM basierenden Austauschs fungieren.
- **Analysis**: Erklärt Aktivitäten die auf die in der Ebene *Resource* angeführten Datenquellen ausgeführt werden. Hierzu zählt u.a. der ETL-Prozess oder das OLAP Modell um Daten in Würfel und Dimensionen aufzubereiten.
- **Management**: Unterstützt das tägliche Geschäft des DW durch die Möglichkeit Aufgaben (engl. Tasks) zu planen und deren zeitliche Durchführung festzulegen. Weiters wird das Aufzeichnen von Aktivitäten des DW sowie das Erstellen von Statistiken unterstützt.

3.7 Zusammenfassung

Durch die Anwendung der modellgetriebenen Architektur entstehen neue Möglichkeiten für den Entwickler. So verlagert sich die Entwicklung von neuer Software auf eine abstraktere und damit plattformunabhängige Ebene. Dadurch wird die Verständlichkeit für fachlich geschulte Experten (nicht Entwickler) erhalten. Weiters wurden in diesem Kapitel die in dieser Arbeit eingesetzten Modellierungsprachen wie UML, CWM und das zugrundeliegende MOF beschrieben.

4 Grundlagen Eclipse

In diesem Kapitel wird zu Beginn auf den geschichtlichen und rechtlichen Hintergrund von Eclipse eingegangen. Anschließend wird die technische Architektur von Eclipse beschrieben. Zum Schluss werden die theoretischen Grundlagen der in dieser Arbeit angewendeten Frameworks erläutert.

4.1 Eclipse und die Eclipse Foundation

Eclipse ist der Name einer Open-Source Gemeinde, welche sich auf die Erstellung einer Entwicklungsumgebung spezialisiert hat. Die Entwicklungsumgebung beinhaltet Erweiterungen und Werkzeuge welche den Entwicklungsprozess von Software in allen Schritten unterstützen. Eclipse wird von der Eclipse Foundation entwickelt und erweitert, welche u.a. die infrastrukturellen Grundlagen (CVS/SVN, Bugzilla, Datenbanken, Mailings Lists, …) für die Open-Source Gemeinschaft bereitstellt. Wichtig ist, dass die Eclipse Foundation einen „not-for-profit“-Charakter hat und darauf abzielt, die Open-Source Gemeinde sowie die Anzahl der Produkte und Services zu erweitern [Eclipse Org 2008].

Definition 4-1: Der Begriff *Open-Source* ist eine Strategie der Softwareentwicklung bei welcher der Quellcode allen Interessierten kostenlos zur Verfügung gestellt wird. Diese werden aufgefordert das Programm zu verbessern und die Ergebnisse wieder der Gemeinschaft bereit zu stellen [Heinrich 2004 S. 473].

Anfänglich wurde Eclipse als internes Projekt von IBM geführt. Im Jahr 2001 wurde es jedoch unter die CPL (Common Public License) gestellt und zur weiteren Entwicklung einem Open-Source Konsortium übergeben. Gründungsmitglieder des Konsortiums waren *Borland*, *IBM*, *MERANT*, *QNX Software Systems*, *Rationale Software*, *Red Hat*, *SuSE*, *TogetherSoft* und *Webgain*. Bis Ende des Jahres 2003 stieg die Anzahl der Mitglieder von den neun Gründungsmitgliedern auf über 80 Mitglieder an [Eclipse Org 2008, Gruhn et al. 2006, S. 279].

Die Führung von Eclipse untersteht dem „Board of Stevens“, bestehend aus einem Repräsentanten pro Mitglied des Konsortiums. Die Repräsentanten überwachen zum einen die Erreichung einer gesunden Ausgeglichenheit zwischen Open-Source- und kommerziellen Interes-

sen aller Mitglieder und zum anderen die technischen Aspekte, wie z.B. die Koordination der Subprojekte [Gruhn et al. 2006, S. 279].

Im Februar 2004 wurde durch das „Board of Stevens" eine Umstrukturierung von Eclipse in eine „not-for-profit" Gesellschaft beschlossen. Dadurch wurde Eclipse eine eigenständige Gesellschaft, deren Ziel es ist den Nutzen der Softwareanbieter sowie der Endbenützer zu maximieren. Alle entwickelten Technologien werden an die Öffentlichkeit freigegeben [Eclipse Org-PR 2008].

4.1.1 Common Public License (CPL) und Eclipse Public License (EPL)

Eclipse hat das Ziel dem Anspruch gerecht werden, kommerziellen als auch Open-Source Anforderungen zu entsprechen. Aus diesem Grund wurde die CPL herangezogen, welche eine Veröffentlichung von auf Eclipse aufbauenden Produkten unter eigener Lizenz ermöglicht. D.h. dass aufbauend auf öffentlichen und freien Frameworks Programme erstellt und unter einer eigenen, möglicherweise kommerziellen Lizenz veröffentlicht werden dürfen. Im Gegensatz zu den meist angewandten Copyright Lizenzen werden diese im Fachbereich Copyleft Lizenzen genannt [Gruhn et al. 2006, S. 280].

Die Eclipse Public License (EPL) ist eine von CPL Version 1.0 abgeleitete Lizenz. Mit der Umstellung von CPL auf EPL gelang es Eclipse einige Unklarheiten und Bedenken zu bereinigen. So wurde im Zuge der Umstellung der Lizenz der Agreement Steward von IBM für CPL auf Eclipse Foundation für EPL geändert. Der Agreement Steward bezeichnet jene juristische oder natürliche Person, welche das Recht hat neue Versionen eines Agreements bzw. einer Lizenz zu veröffentlichen. Dies macht das Agieren der Foundation von IBM unabhängiger. Hierzu wurde als einzige Änderung folgender Satz entfernt:

> *If Recipient institutes patent litigation against a Contributor with respect to a patent applicable to software (including a cross-claim or counterclaim in a lawsuit), then any patent licenses granted by that Contributor to such Recipient under this Agreement shall terminate as of the date such litigation is filed.* [Eclipse EPL-FAQ 2009]

Diese Klausel ermöglichte es dem Patentgeber nachträglich noch gesetzliche Schritte gegenüber der in Anspruch nehmenden Partei einzuleiten. Durch das Entfernen dieser Klausel er-

hoffen sich die Mitglieder der Eclipse Foundation die Größe des ökologischen Systems rund um Eclipse zu erweitern.

4.2 Architektur

Die Architektur der Eclipse Rich-Client-Plattform charakterisiert sich durch den modularen Aufbau. Im Zentrum des Projekts steht ein kleiner fester Kernel mit den unterstützenden Diensten und Komponenten der Eclipse Plattform. Alle Funktionalitäten bzw. konkreten Anwendungen werden in Form von Erweiterungen bzw. Plug-ins angefügt. So wurden z.B. auch die Java Entwicklungsumgebung oder der Debugging-Modus als Plug-in umgesetzt. Die Anbindung erfolgt über vom Kernel bereitgestellte Erweiterungspunkte (engl. Extension Points). Die angebundenen Plug-ins können wiederum Erweiterungspunkte für andere Anwendungen bereitstellen [Gruhn et al. 2006 S. 280f].

Definition 4-2: Ein *Plug-in* ist ein Erweiterungsmodul für bestimmte Softwareanwendungen. Die Anbindung erfolgt durch, vom Grundsystem bereitgestellte, Schnittstellen. Die Möglichkeit Plug-ins anzubinden deutet auf eine modulare Struktur des Grundsystems hin.

Aufgrund der modularen Struktur werden Eclipse oftmals die Grundlagen für eine allgemeine Plattform für Desktop- und Rich-Client-Anwendung nachgesagt. Seit Version 3.0 kann Eclipse auch als eine eigenständige Plattform für die Erstellung von Rich-Client-Plattformen eingesetzt werden. Hierzu waren eine große Menge an Anpassungen der verschiedenen Plug-ins der jeweiligen Herstellern notwendig [Daum 2005 S. 25f].

Definition 4-3: Ein *Rich-Client* ist eine Anwendung durch welche die gängigen grafischen Darstellungsformen eines Betriebssystems adaptiert bzw. verwendet werden können. Somit koppelt sich der Rich-Client nicht von der Erscheinung des Systems ab. Eine weitere grundlegende Eigenschaft ist die Verbindung zum Server bzw. die Aktivitäten der Anwendung bei nicht vorhandener Verbindung. Hierbei muss der Rich-Client immer noch ausführbar sein. Sobald die Verbindung wieder vorhanden ist, sollten die Daten mit dem Web abgeglichen werden. Konfiguration und Aktualisierung der Anwendung sollten über das Web und ohne Benutzerinteraktion erfolgen [Daum 2005 S. 27f].

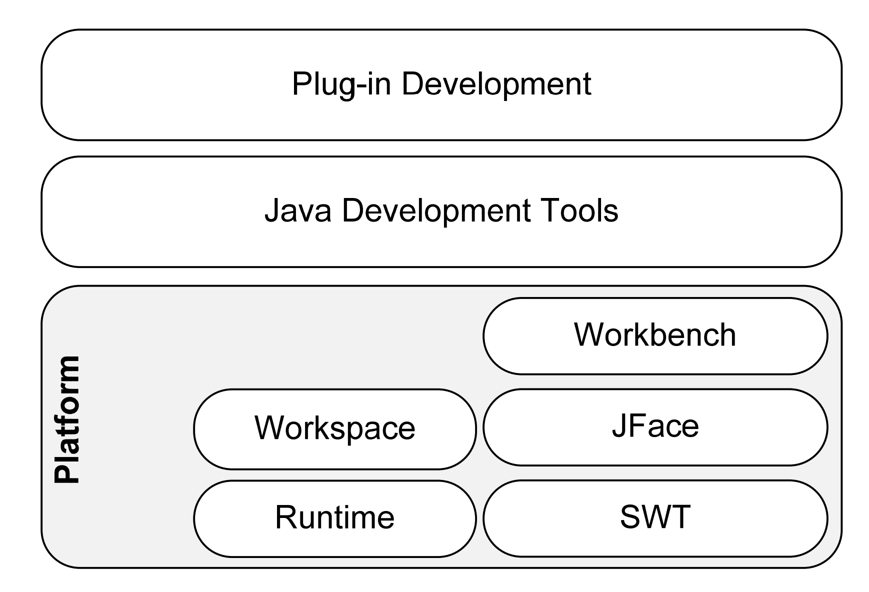


Abbildung 4-1: Architektur Eclipse Distribution (basierend auf [Gruhn et al. 2006 S. 281])

Abbildung 4-1 stellt die drei Schichten der Architektur einer Standard Eclipse Distribution dar. Besonders hervorzuheben ist die Runtime-Komponente in der Plattformschicht. Mittels dieser Komponente werden beim Start der Eclipse Plattform die notwendigen Metadaten für die jeweiligen angebundenen Plug-ins eingelesen. Das komplette Plug-in wird erst bei der tatsächlichen Verwendung „*just-in-time*“ nachgeladen. Würden alle Plug-ins beim Start in die Plattform eingelesen werden, würde dies die Startzeit der Plattform immens beeinträchtigen [Gruhn et al. 2006 S. 281ff].

Die Runtime-Komponente benützt als Ablaufumgebung die Dienste des OSGi (Open Service Gateway Initiative)-Standards. Durch den Einsatz dieser Technologie vereinfacht sich die Verwaltung der verwendeten Komponenten. Es wird somit eine Standardisierung der Dienste angestrebt. OSGi konforme Dienste können ausschließlich auf OSGi konformen Servern ausgeführt werden [Daum 2005 S. 26].

Die für die Darstellung der Oberfläche des Programms verantwortlichen Komponenten sind auf der rechten Seite der Abbildung 4-1 angeführt. Durch die Workbench werden die grundlegenden Elemente von Eclipse wie Editoren, Sichten und Perspektiven definiert. Diese werden bei der Entwicklung durch Funktionalitäten erweitert. SWT ist ein Framework, welches eine Anpassung der Oberfläche an das Betriebssystem ermöglicht. Eine genauere Beschreibung dieses Frameworks erfolgt im Kapitel über die Implementierung des Werkzeuges, siehe Abschnitt 6.4.5. JFace baut auf SWT auf und stellt Standardimplementierungen für oft ver-

wendete grafische Elemente wie z.B. Assistenten (engl. Wizard) oder Dialoge, bereit. Im weiteren Verlauf wird oftmals der englische Begriff *Wizard* anstelle des Begriffes Assistent verwendet. Durch den englischen Begriff können einige Beispiele deutlicher erklärt werden.

Weitere Bestandteile der Standarddistribution von Eclipse sind die Java Development Tools welche aus dem das Java Development Toolkit und Source Debugger Plug-in bestehen. Zusätzlich wird noch das Plug-in Development Environment bereitgestellt. [Gruhn et al. 2006 S. 281ff]

4.3 Grundlagen eines Eclipse Plug-ins

In diesem Kapitel wird das grundlegende Wissen für die Entwicklung eines Plug-ins in der Rich-Client-Plattform Eclipse bereitgestellt.

Als Entwicklungssprache für Eclipse Plug-ins wird die Programmiersprache Java vorgegeben. Ein Plug-in besteht demnach in der Regel aus einem oder mehreren Java-Archiven. Die notwendigen Elemente eines Plug-ins werden in Eclipse durch den dafür passenden Assistenten sofort bei der Erstellung erzeugt. Aufgabe des Benutzers ist es diese zu verwalten.

Herzstück eines jeden Plug-ins ist ein Manifest, in welchem die Konfiguration eines Plug-ins im Detail beschrieben wird. Dieses Plug-in stellt die Metadaten für den Start der Eclipse-Plattform bereit, d.h. nur diese Daten werden zu Beginn geladen. Die dazugehörigen Java-Dateien werden erst bei Bedarf nachgeladen. Die Umsetzung des Manifests wird mittels der XML Datei *plugin.xml* sowie der Datei *META-INF/MANIFEST.MF* durchgeführt.

Für das Editieren der beiden Dateien steht ein Editor bereit, der die notwendigen Schritte unterstützt. In Listing 4-1 wird eine beispielhafte Implementierung einer MANIFESTS.MF Datei dargestellt, wobei diese nur auf wichtigsten Elemente reduziert wurde. Die ersten beiden Zeilen stellen klar, dass es sich um eine OSGi Manifest Datei handelt. In den nachstehenden Zeilen werden u.a. Plug-in-Name, Version, Pfad und Aktivator angegeben. Da in dieser Datei viele komplexe Pfade angewandt werden, empfiehlt es sich für die Umsetzung die Unterstützung des dafür vorgesehenen Editors in Anspruch zu nehmen. [Gamma et al. 2006]

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Example Plugin
Bundle-SymbolicName: org.eclipse.uml2.uml.editor; singleton:=true
Bundle-Version: 2.2.0.v200805131030
Bundle-ClassPath: org.eclipse.uml2.uml.editor_2.2.0.v200805131030.jar
Bundle-Activator: org.eclipse.uml2.uml.editor.UMLEditorPlugin$Implementation
…
```

Listing 4-1: Beispiels MANIFEST.MF Datei

Im Manifest werden auch die verschiedenen Erweiterungsmechanismen definiert. So wird zum einen bestimmt, welches Plug-in durch das entwickelte erweitert wird und welche Erweiterungsmöglichkeiten anderen Plug-ins bereitgestellt werden. Dies erfolgt mithilfe der sogenannten *Extensions* und *Extension Points*. [Daum 2005 S. 25f]

In Listing 4-2 wird eine Implementierung der Datei *plugin.xml* dargestellt. In diesem Beispiel wird eine einzelne Erweiterung (engl. Extension) angefügt. Es lässt sich durch den Erweiterungspunkt (engl. Extension Point) org.eclipse.ui.newWizards, als auch durch den Tag <*wizard*> erkennen, dass die Erweiterung in Form eines *Wizards* umgesetzt wurde. Der Tag <*category*> beschreibt die Einordnung des *Wizards* in der Anzeige von Eclipse, hierbei kann entweder eine schon vorhandene oder eine neue Kategorie angegeben werden. In der Definition des *Wizards* wird u.a. eine Referenz auf die zuvor definierte Kategorie erzeugt. Darüberhinaus wird die bei Aktivierung des *Wizards* aufzurufende Klasse, das dazugehörige Bild, die Id und die Bezeichnung angegeben. Für die Datei *plugin.xml* gilt bezüglich des Editors dasselbe wie für die Datei *MANIFEST.MF*, da die Bearbeitung durch den Editor um vieles vereinfacht wird [Gamma et al. 2006].

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
  <extension
     point = "org.eclipse.ui.newWizards">
    <category
          id="org.eclipse.uml2.uml.editor"
          name="Sample Wizards">
    </category>
    <wizard
          category="org.eclipse.uml2.uml.editor"
          class="org.eclipse.uml2.uml.editor.wizards.SampleNewWizard"
          icon="icons/sample.gif"
          id="org.eclipse.uml2.uml.editor.wizards.SampleNewWizard"
          name="Multi-page Editor file">
    </wizard>
  </extension>
</plugin>
```

Listing 4-2: Beispiel für plugin.xml Datei

4.4 Eclipse Tool Project

Das Eclipse Tool Project ist ein Unterprojekt des allgemeinen Eclipse Projekts. Dieses Projekt konzentriert sich schwerpunktmäßig auf Applikationen welche den Entwicklern von Plug-ins die Arbeit erleichtern. Hierzu zählen u.a. Werkzeuge und Frameworks welche z.B. die Code-generierung bzw. auch die Erstellung von visuellen Editoren unterstützen.

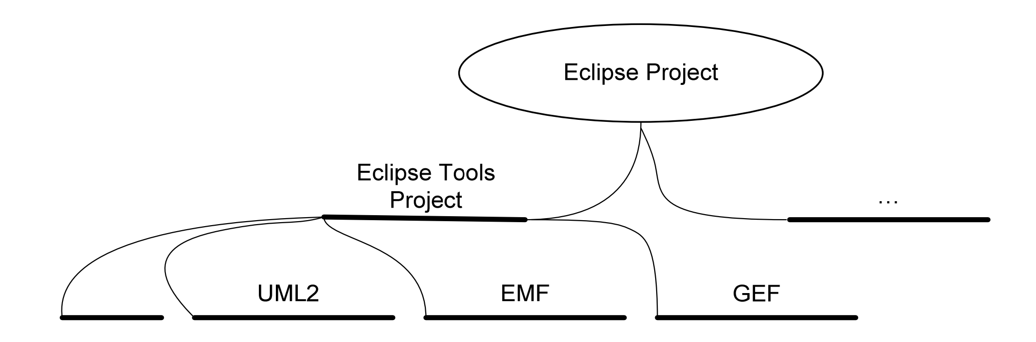


Abbildung 4-2: Übersicht über Eclipse Tools Project (basierend auf [Gruhn et al. 2006 S. 282]

Die in Abbildung 4-2 angeführten Frameworks werden in den folgenden Unterpunkten im Detail beschrieben. Der Einsatz dieser Frameworks ist in dieser Arbeit dezidiert vorgegeben, daraus ergibt sich die Notwendigkeit einer detaillierten Beschreibung.

Die wirtschaftliche Relevanz des Eclipse Tools Projects ergibt sich speziell aus den rechtlichen Rahmenbedingungen, da der Großteil der Unterprojekte unter der CPL bzw. der EPL veröffentlicht wurde und somit eine kommerzielle Nutzung ermöglicht.

4.4.1 Eclipse Modeling Framework

In der Entwicklung eines Programms können verschiedene Sichten auf die Problematik gefordert werden. Zur Definition eines einfachen Tatbestands kann für einen geübten Java-Entwickler als Datenmodell der dazu passende Quelltext völlig ausreichend sein. Für nicht geschulte Entwickler oder wirtschaftliche Fachkräfte ist dies jedoch zu wenig, da sie aus dem reinen Code keine Information lesen können. Hierfür bieten sich UML-Diagramme (siehe Abschnitt 3.5) an, da diese leicht anpassbar und verständlich sind. Zusätzlich befriedigen sie den Wunsch von Unternehmen nach einer gründlichen Dokumentation. Für eine konsistente Speicherung des Datenmodells, welches bereits in UML und als Quelltext abgebildet ist, muss

wiederum ein neues Format gewählt werden. Eine Umsetzung wäre z.B. mit einem XML Dokument möglich.

Gemäß der oben beschriebenen Problematik muss dasselbe Datenmodell auf drei verschiedene Arten implementiert werden. Noch viel mehr muss der Entwickler ein Spezialist in den verschiedenen Bereichen sein, um der Aufgabenstellung nachkommen zu können. Genau an dieser Problematik setzt EMF an. Durch EMF können die verschiedenen Datenmodelle auf eine geordnete Abbildung zusammengefasst werden. Abbildung 4-3 illustriert den zentralen Charakter des EMF. [Budinsky et al. 2004 S. 9ff]

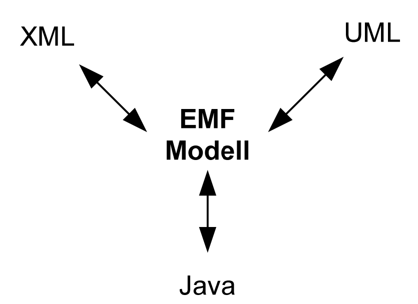


Abbildung 4-3: EMF vereinigt Java, XML und UML (basierend auf [Budinsky et al. 2004 S. 12])

Ein EMF-Modell kann auf verschiedene Arten erstellt werden. So besteht z.B. die Möglichkeit ein UML-Diagramm zu zeichnen und daraus das EMF-Modell zu generieren. Die Generierung von Elementen ist ein zentrales Thema des EMF. Aus einem generierten EMF-Modell kann auch XML und Java Code erzeugt werden. Für den Entwickler ergibt sich dadurch der Vorteil, dass er das Datenmodell nur einmal erstellen muss und den Rest daraus generieren kann. Speziell bei nachträglichen Änderungen ist dies von großem Vorteil, da diese automatisiert in die verschiedenen Repräsentationen übernommen werden können.

EMF steht somit zwischen den Anwendern und den Gegner der MDA (siehe Abschnitt 3.2). Es wird versucht, die Vorteile aus beiden Welten zu vereinen und so ein möglichst umfassendes Framework zur Erstellung von Anwendungen zu geben.

Das Ecore-Model

Mithilfe des Ecore-Models werden die EMF-Modelle beschrieben. Das Ecore-Model kann als eine Implementierung des EMOF (siehe Abschnitt 3.4) betrachtet werden. Eine Implementierung des CMOF ist aufgrund der hohen Komplexität (noch) nicht umsetzbar. Das Ecore-

Model ist somit der Meta²-Modell Ebene zuzuordnen. Die wichtigsten Bestandteile des Ecore-Models, der Kernel, werden in folgender Grafik beschrieben.

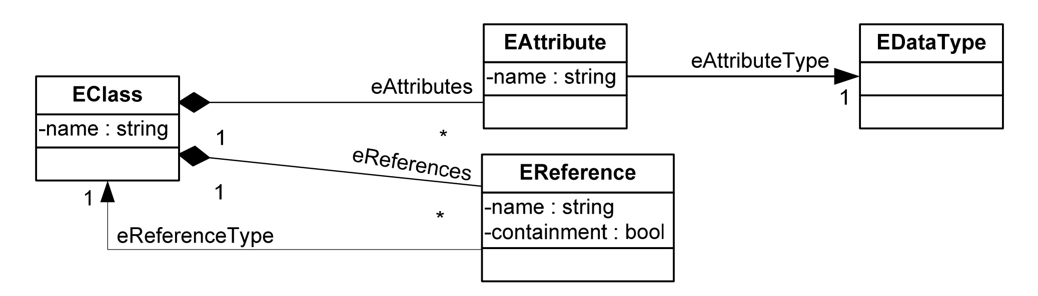


Abbildung 4-4: Vereinfachte Darstellung des EMF-Models (basierend auf Budinsky et al. 2004 S. 16])

Wie man in Abbildung 4-4 erkennen kann besteht der Kernel des Ecore-Modells aus vier Klassen [Budinsky et al. 2004 S. 16]:

- **EClass**: Wird verwendet um die modellierte Klasse darzustellen. Beim Erstellen der Klasse muss verpflichtend ein Name angegeben werden. Sie kann optional Attribute und Referenzen beinhalten.
- **EAttribute**: Stellt ein Attribut dar. Beinhaltet einen Namen und muss einem Typ entsprechen.
- **EReference**: Repräsentiert das Ende einer Assoziation zwischen verschiedenen Klassen. Es beinhaltet einen Namen, sowie einen booleschen Wert, der überprüft ob die Referenz rekursiv ist. Zusätzlich wird das Zielattribut referenziert.
- **EDataType**: Gibt den Datentyp eines Attributes an, der sowohl primitiv als auch ein Objekt sein kann.

Die Architektur des Ecore-Modells stellt einen Teil der UML-Spezifikation dar. Eine Umsetzung des kompletten UML-Metamodells ist in EMF noch nicht möglich.

4.4.2 UML2-Framework

Das UML2-Framework ist eine auf EMF basierende Implementierung des UML (siehe 3.5) Metamodells für die Eclipse Plattform. Dabei werden folgende grundlegende Ziele verfolgt:

- Die Erstellung einer benützbaren Implementierung des UML-Metamodells, um damit die Entwicklung von Modellierungswerkzeugen zu unterstützen.

- Ein einheitliches XMI Schema, um damit den Austausch von semantischen Modellen zu vereinfachen.
- Das Festlegen von Validierungsregeln, um damit Fehler frühzeitig zu erkennen. [Eclipse UML2 2009]

Das UML2-Framework bietet u.a. die Möglichkeit den Code aus dem erstellten Modell zu erzeugen. Hierfür werden die Codegenerierungsmechanismen von EMF verwendet. So kann aus einem UML-Modell, ein Ecore- und ein Generator-Model erzeugt werden, aus welchen in weiterer Folge der notwendige Code generiert wird [Eclipse UML2 FAQ 2009].

UML2 kommt mittlerweile in einer Vielzahl von verschiedenen Werkzeugen zum Einsatz. Dies ergibt sich u.a. daraus, dass UML2 auch „Standalone“, also ohne Eclipse ausgeführt werden kann. Es müssen lediglich die notwendigen Klassenbibliotheken angebunden sein. Zu den kommerziellen Werkzeugen welche UML2 einsetzen zählen *Omondo EclipseUML*, *IBM RSM/RSA*, *MagicDraw UML* und *Soyatect eUML2*. Zusätzlich gibt es eine Vielzahl an verschieden unter der EPL entwickelten Werkzeuge wie z.B. *Papyrus*, *Taylor MDA*, *MOSkitt* oder die in dieser Arbeit verwendeten *Eclipse UML2 Tools*. Wie man am großen Angebot der verschiedenen Werkzeuge erkennen kann, ist die praktische und wirtschaftliche Nachfrage nach dem UML2-Framework sehr hoch [Eclipse UML2 Co. 2009]. Eine Referenzliste mit den Links zu den oben beschriebenen Programmen findet sich unter [Eclipse UML2 Co. 2009].

Eclipse UML2 Tools

Die UML2 Tools stellen Editoren zu den wichtigsten grundlegenden UML-Diagrammen bereit. Hierzu zählen u.a. das Klassendiagramm, das Aktivitätsdiagramm, das Anwendungsfalldiagramm sowie das Sequenzdiagramm. Eine genauere Beschreibung der in diesem Projekt verwendeten Diagramme findet sich unter Kapitel 3.5. [Eclipse UML2 Tools 2009]

Die Repräsentation eines UML-Diagramms erfolgt über zwei verschiedene Dateien. Einer Modell-Datei (**.uml*) welche die Daten beinhaltet und die Grundlage für die darauf aufbauende Diagramm-Datei (**.umlclass*) darstellt. [Eclipse UML2 T. FAQ 2009]

Die Integration der Eclipse UML2 Tools erfolgt, falls nicht in der verwendeten Eclipse Distribution vorhanden sein sollten, über den von Eclipse bereitgestellten Update Manager. Auf

die konkrete Anwendung des UML2 Frameworks sowie der UML2Tools und deren Patterns und Designkonzepten wird nochmals spezifisch in Kapitel 6 eingegangen.

4.4.3 Graphical Editing Framework (GEF)

GEF ist ein Framework welches es dem Entwickler erlaubt, aus einem bereits bestehenden Datenmodell einen grafischen Editor zu erstellen. Der Entwickler profitiert aus einer Menge an bereits vorgefertigten Operationen und Funktionalitäten zur Umsetzung von grafischen Elementen und Interaktionen zwischen diesen. Gegebenenfalls können diese durch den Benutzer erweitert werden [Eclipse GEF 2009].

Die Umsetzung des GEF-Frameworks baut auf dem Draw2d Framework auf. Draw2d ist für die Repräsentation von Daten entwickelt worden. Dies bedeutet, dass keine Funktionen damit angewandt werden können. Genau an diesem Punkt setzt GEF an und gibt dem Benutzer bzw. dem Entwickler die Möglichkeit auf einfache Art und Weise z.B. „*Drag & Drop*“ Funktionen umzusetzen. Der Entwickler kann das Verhalten von grafischen Elementen durch Setzen von Eigenschaften definieren. So ist es z.B. möglich einem Element die Eigenschaft „ziehbar“ zu zuweisen.

GEF ist auf Basis des MVC-Frameworks (siehe Abschnitt 7.1.4) umgesetzt, was eine strikte Trennung der verschiedenen Komponenten unterstützt und somit dem Entwickler einfache Möglichkeiten zu Erweiterung des Editors gibt. Darüberhinaus wird die Wiederverwendbarkeit des Codes gewährleistet, da z.B. eine grafische Repräsentation nicht direkt an ein Element gebunden ist und somit auf ein anderes Element erneut angewendet werden kann. [Eclipse GEF FAQ 2009]

4.5 Zusammenfassung

In diesem Kapitel wurden die grundlegenden Konzepte von Eclipse, der Erstellung eines Plug-ins sowie die in dieser Arbeit relevanten Frameworks auf theoretischer Basis erläutert. Die Frameworks wurden anhand ihrer Grundgedanken bzw. die Einsatzgebiete beschrieben. Die technologischen Eigenschaften der jeweiligen Frameworks werden in Kapitel 6 Entwicklung des Global Schema Architect dezidiert beschrieben.

II. Umsetzung des Werkzeuges

5 Erstellung des Metamodells

Grundlage für die Umsetzung des Werkzeuges ist ein Metamodell auf welchem basierend in Eclipse ein EMF-Modell umgesetzt wird. Ein Metamodell ermöglicht es die unterschiedlichen lokalen Data Mart Schemata einheitlich zu Repräsentieren. Darauf aufbauend können die Data Marts strukturiert miteinander verglichen werden. Zu Beginn werden die Anforderungen an das Metamodell besprochen. In weiterer Folge werden andere Metamodelle mit ähnlichen Zielen verglichen und darauf aufbauend wird schlussendlich das verwendete Metamodell erstellt.

5.1 Grundlegende Anforderungen an das Metamodell

Data Marts werden durch das multidimensionale Datenmodell dargestellt. In einem föderierten DW-System müssen Data Marts aus verschiedenen Quellen einheitlich abgebildet werden um einen Vergleich zu ermöglichen. Aus diesem Grund ist die Erstellung eines einheitlichen Metamodells notwendig. Das Metamodell bildet in weiterer Folge die Grundlage für die Umsetzung der Schemata in der Implementierung des Plug-ins. Folgende Anforderungen werden an das Metamodell gestellt.

5.1.1 Darstellung des multidimensionalen Datenmodells

In 2.1.4 wurde das multidimensionale Datenmodell detailiert beschrieben. An das Metamodell wird die Forderung gestellt alle im multidimensionalen Datenmodell ausdrückbaren Konstrukte zu unterstützen.

5.1.2 Darstellung in UML

Durch UML als Darstellungsform soll ein hoher Wiedererkennungswert beim Benutzer erreicht werden, wodurch die Einarbeitungsphase verkürzt wird. Die am besten geeignetste Darstellungsform ist das Klassendiagramm, weil damit statische Zusammenhänge zwischen verschiedenen Elementen (wie z.B. Dimension und Fakten) ausgedrückt werden können.

5.1.3 Übersichtliche Darstellung

Das Werkzeug soll so übersichtlich und benutzerfreundlich wie möglich aufgebaut sein. Um ein Model verständlich und nachvollziehbar darzustellen, ist es wichtig klare Regeln und

Rahmenbedingungen festzulegen. Aus diesem Grund ist ein strukturiertes Metamodell notwendig.

5.1.4 Konformität zu anderen Werkzeugen

Durch die geforderte Interaktion des Modells mit anderen Modellen ist die Verwendung von Standards notwendig. In der Umsetzung dieses Werkzeuges muss Kompatibilität zu dem im SQL-MDi Query Processor und SQL-MDi Query Parser verwendeten Datenmodell gegeben sein. Zu den zu verwendenden Standards zählen das CWM sowie die in 5.2 beschriebenen Metamodelle.

5.2 Data Warehouse Metamodelle

In der Literatur finden sich zahlreiche Modellierungsansätze für DWs. Folgende werden aufgrund von vielen Referenzen in anderen wissenschaftlichen Arbeiten näher betrachtet.

5.2.1 The unified multidimensional metamodel

[Akoka et al. 2006 S. 1450ff] versucht mehrere Ansätze der Literatur zu einem einheitlichen Metamodell zusammenzuführen. Das Metamodell muss unabhängig von jeglicher Implementierung entwickelt werden und demnach auf jede Entwicklung angewendet werden können.

Zu den Kernkonstrukten des Modells zählen Dimensionen und Fakten, welche in Abbildung 1-1 fett herausgehoben werden. Dimensionen werden durch in Hierarchien abgelegte Aggregationsebenen (engl. *Dimensionlevel*) beschrieben. *ClassificationRelationships* speichern die Beziehungen der verschiedenen Aggregationsebenen zueinander und definieren somit die Hierarchie einer Dimension. Basierend auf der Hierarchie können in weiterer Folge Roll-up und Drill-down Operationen ausgeführt werden. Eine Hierarchie besteht demnach aus einer Menge von *ClassificationRelationships*. Eine Aggregationsebene wird durch eine Menge an Attributen definiert. Diese unterteilen sich in identifizierende (engl. *identifying*) und nicht identifizierende (engl. *not identifying*) Attribute.

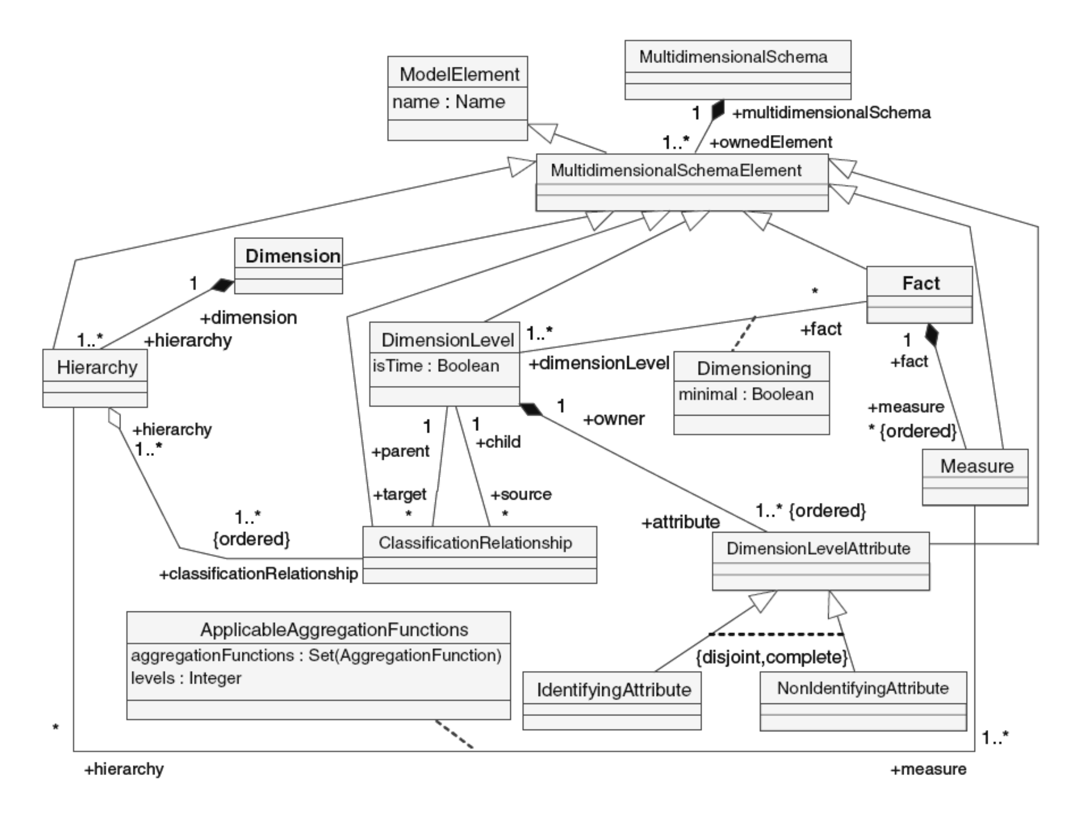


Abbildung 5-1: Unified multidimensional metamodel [Akoka et al. 2006 S. 1453]

Fakten (engl. *Fact*) bestehen aus verschiedenen Kenngrößen (engl. *Measures*) und werden durch referenzierte Aggregationsebenen näher beschrieben. In diesem Ansatz kann zwischen minimalen und nicht minimalen Aggregationsebenen unterschieden werden. Als Beispiel für diese Problematik kann man den Verkauf eines Autos heranziehen. Der Verkauf eines Autos wird durch die Dimensionen Kunde, Fahrzeug und Tag näher beschrieben. Für die Identifizierung eines Verkaufs sind Fahrzeug und Tag ausreichend. Die Information an wem das Auto verkauft wurde, ist demnach nicht unbedingt notwendig und somit nicht minimal. In diesem Beispiel ist es möglich verschiedene Dimensionen mehrmals verwenden und unterschiedlichen Fakten zu zuordnen [Akoka et al. 2006 S. 1450ff].

Durch die Einführung der Aggregationsfunktionen (engl. *AggregationsFunctions*) können auch nicht numerische Werte in einem Fakt verwendet werden. Für die einzusetzenden Werte müssen Aggregationsfunktionen, wie z.B. SUM, AVG, MIN oder COUNT, implementiert worden sein. Für numerische Werte sind diese Funktionen meist standardmäßig von der Implementierungsumgebung vorgegeben.

Zusammengefasst kann man diesen Ansatz als vollständig angesehen. Bei großen Modellen verliert er jedoch deutlich an Übersichtlichkeit und Lesbarkeit. Abbildung 5-2 zeigt diese Problematik. Schon bei relativ geringer Komplexität wird das Modell schwer überschaubar.

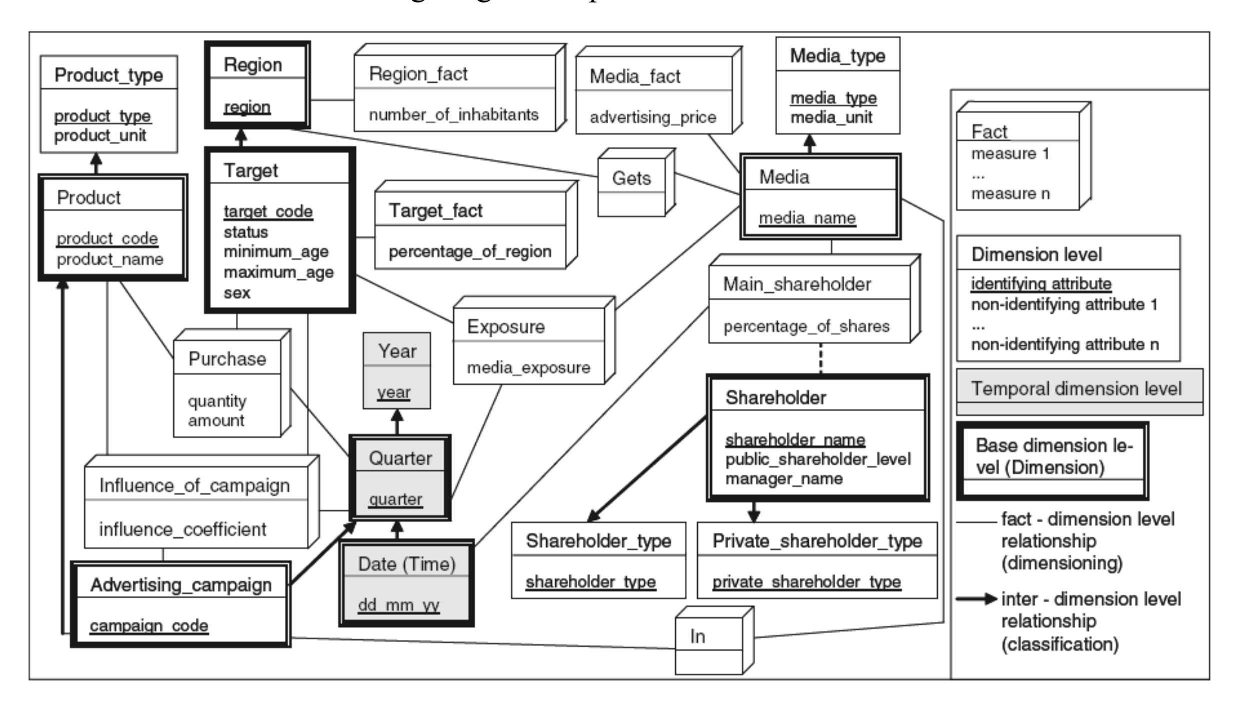


Abbildung 5-2: Beispiel für Unified Multidimensional Metamodel [Akoka et al. 2006 S. 1466]

5.2.2 Modellierung mit UML Package Diagrammen

Dieser Ansatz versucht primär die Darstellung eines multidimensionalen Modells zu vereinfachen. Zusätzlich wird versucht nahe am Gedankenmodell des Entwicklers und des Analysten zu bleiben. Um dies zu erreichen wurde das Metamodell in drei hierarchische angeordnete Ebenen (engl. *Level*) unterteilt, welche mittels Paketen dargestellt werden. [Luján-Mora et al. 2002]

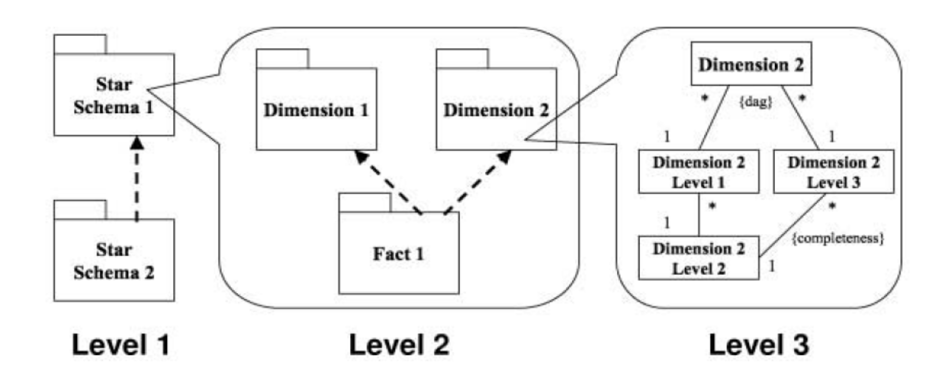


Abbildung 5-3: Die drei Ebenen von multidimensionalen Modellen [Luján-Mora et al. 2002 S. 204]

- **Ebene 1**: Auf der ersten Ebene werden die Würfel (z.B. Star Schema) angeführt. Die Repräsentation eines Würfels erfolgt über ein Paket. Die Verbindungen zwischen den Würfeln werden mit Pfeilen bzw. sogenannten *Dependencies* dargestellt. Eine Verbindung zwischen zwei Star-Schemas zeigt an, dass zumindest eine Dimension geteilt wird. Eine Verknüpfung kann deshalb nur aus Abhängigkeiten der im Schema beinhalteten Elemente erfolgen.
- **Ebene 2**: Diese Ebene stellt Dimensionen als auch Fakten und übergreifenden Beziehungen dar. Die Darstellung sowohl der Dimensionen als auch der Fakten erfolgt über die UML-Darstellung eines Pakets. Bei Verbindungen zwischen Dimensionen wird davon ausgegangen, dass zumindest eine Ebene in beiden Würfeln vorkommt.
- **Ebene 3**: Die genaue Struktur der Dimensionen und Fakten wird in Form von Klassen detailliert dargestellt. Fakten werden durch einzelne Attribute und Dimensionen durch eine Hierarchie von Attributen repräsentiert. In der Repräsentation von Fakten sind daher keine Verbindungen zwischen den einzelnen Attributen gegeben [Luján-Mora et al. 2002 S. 204]

Dieses DW Metamodell beschränkt sich auf die Teilung in drei verschiedene Modellebenen. Zufolge des Autors sind zu tiefe Hierarchien verwirrend und in diesem Beispiel nicht notwendig [Luján-Mora et al. 2002 S. 204].

Zusammenfassend liefert dieses Modell eine einfache Variante um komplexe Modelle in leichter lesbare Einzelteile zu spalten. Folgende Abbildung hebt die Stärke des Metamodells heraus. Die Aufteilung des gesamten Modells in einzelne Elemente ermöglicht dem Leser das Modell Schritt für Schritt und somit einfacher aufzufassen.

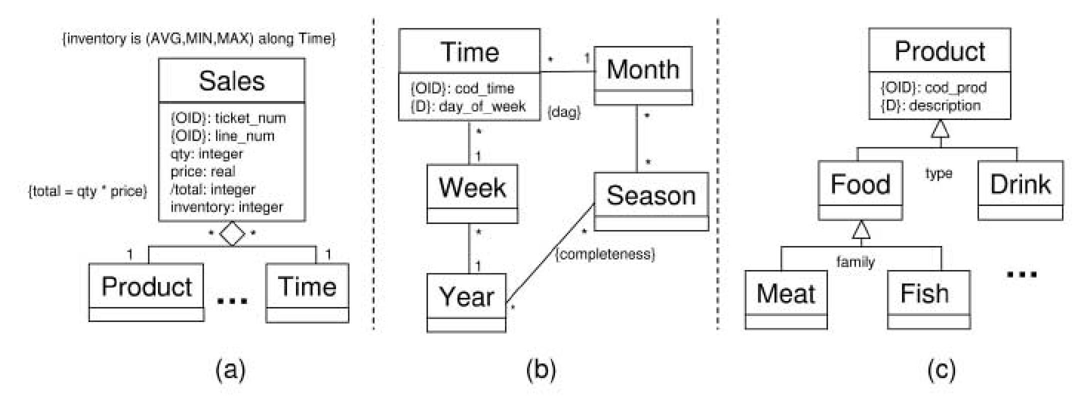


Abbildung 5-4: Beispiel paktetorientierter Ansatz [Luján-Mora et al. 2002 S. 203]

5.2.3 Vergleich der DW Metamodelle

Der Vergleich der DW-Metamodelle erfolgt durch Evaluierung der in 5.1 beschriebenen Anforderungen. Ein objektiver Vergleich der DW-Metamodelle wird anhand einer Prüfung der Abbildbarkeit der einzelnen multidimensionalen Elemente durchgeführt. In Tabelle 5-1 werden die zur Kompatibilität mit den SQL-MDi Query Parser und dem SQL-MDi Query Processor notwendigen multidimensionalen Elemente angeführt.

Multidimensionale Elemente	The unified multidimensional meta-model (1)	Modellierung mit UML Package Diagrammen (2)
Würfel	✓	✓
Dimension	✓	✓
Fakt	✓	✓
Kenngröße	✓	✓
Aggregationsebene	✓	✓
Abbildung Hierarchie	✓	✓
Dimensionsattribut	✓	
Identifizierend vs. Nicht identifizierend	✓	
Wiederverwendbarkeit von Attributen*	✓	✓

Tabelle 5-1: Vergleich multidimensionale Elemente der DW-Modelle

In Tabelle 5-1 wird ersichtlich, dass beide Modelle ausgereift sind und einen Großteil der Anforderungen erfüllen. Modell (2) kann im Vergleich zu Modell (1) keine Dimensionsattribute darstellen. Dies ist jedoch für die Umsetzung des Werkzeuges notwendig. Aus diesem Grund kann Modell (2) nicht vollständig für die Entwicklung des Werkzeuges eingesetzt werden.

In weiterer Folge müssen auch Anforderungen an die Repräsentation der Modelle verglichen werden. In diesem Zusammenhang wird die Umsetzbarkeit in UML geprüft und schlussendlich wird der Aufbau der Modelle aufgrund der Übersichtlichkeit textuell evaluiert.

Anforderungen	The unified multidimensional meta-model (1)	Modellierung mit UML Package Diagrammen (2)
Darstellbarkeit in UML	✓	✓
Übersichtlichkeit	Komplexe Modelle werden deutlich unübersichtlicher. (siehe Abbildung 5-2)	✓ Paketstruktur schafft übersichtliche Gliederung auch bei komplexen Modellen. (siehe Abbildung 5-4)

Tabelle 5-2: Vergleich der Anforderungen an die Repräsentation

Die Darstellung von Modell (1) verliert mit steigender Anzahl an Elementen an Übersichtlichkeit gegenüber Modell (2). Alle an das Metamodell gestellten Anforderungen müssen erfüllt werden, demzufolge ist keines der beiden Modelle für sich allein passend. Für die Umsetzung des GSA-Metamodells müssen die Vorteile der beiden Modelle kombiniert werden.

5.3 Umsetzung des GSA-Metamodells

Keines der in 5.2 dargestellten Metamodelle kann den gestellten Forderungen gerecht werden. Aus diesem Grund muss ein neues Metamodell, das GSA-Metamodell, erstellt werden. Für die Umsetzung werden aus den angeführten Modellen die passenden Konstrukte übernommen.

Um die geforderte Übersichtlichkeit zu erreichen, werden die Konzepte des Ansatzes von [Luján-Mora et al. 2002] als Grundlage herangezogen. Weitere Vorteile dieses Ansatzes sind der einfache Aufbau und die standardmäßige Repräsentation in UML. Aufgrund der beschränkten Anzahl an darstellbaren multidimensionalen Elementen muss der Ansatz um die fehlenden Elemente erweitert werden. Dazu wird der Ansatz von [Akoka et al. 2006] hinzugezogen, aus welchem die für die Konstruktionen dieses Projekts notwendigen Elemente entnommen werden. Um die Kompatibilität mit dem CWM aufrecht zu erhalten wird in erste Linie auf das *Core* Paket des Modells eingegangen.

Zusätzlich muss schon bei der Erstellung des Metamodells auf die von Eclipse vorgegebenen Beschränkungen und Möglichkeiten Rücksicht genommen werden. Das Metamodell muss plattformunabhängig aufgebaut und daraus folgend auch mit den von Eclipse gegebenen Mitteln und Werkzeugen umsetzbar sein.

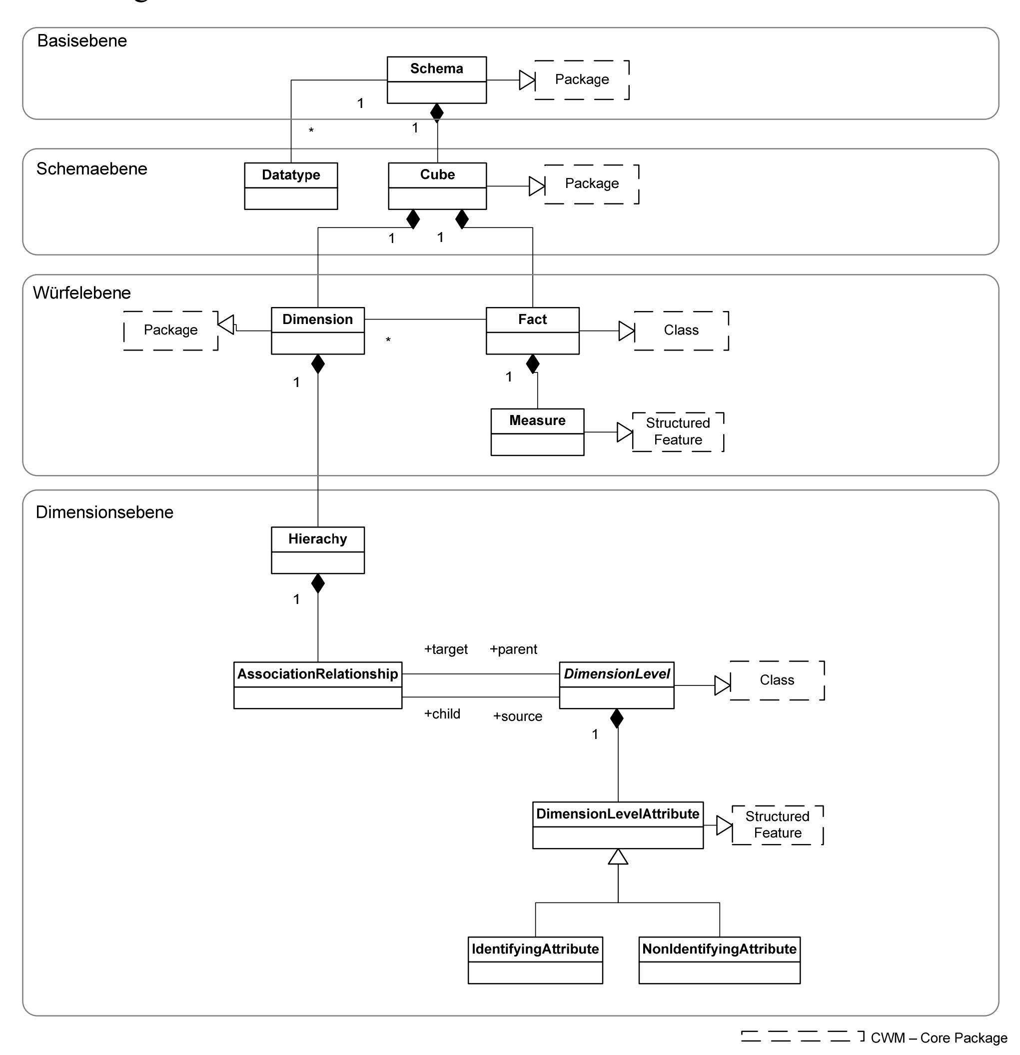


Abbildung 5-5: Metamodell des Werkzeuges

In Abbildung 5-5 wird das in diesem Werkzeug eingesetzte Metamodell illustriert. Im Vergleich zum Metamodell von [Luján-Mora et al. 2002] wird in diesem Ansatz eine weitere Abstraktionsebene, die *Basisebene*, eingeführt. Die Basisebene wird dem Ansatz von [Luján-Mora et al. 2002] vorgelagert um Beziehungen zwischen verschiedenen Würfel besser darstellen zu können. Die Einführung dieser Ebene rechtfertigt sich insbesondere durch die bessere Umsetzbarkeit in EMF, da dadurch Schemata explizit gespeichert werden können.

Folgend wird das Metamodell anhand der eingeteilten Ebenen näher beschrieben. Dazu wird von der abstraktesten zur konkretesten Ebene vorgegangen:

1. **Basisebene**: Stellt die oberste Ebene dar und beinhaltet die verschiedenen Schemata des Projekts. Jedes Schema steht für sich allein, d.h. es können keine Beziehungen zwischen den unterschiedlichen Schemata gesetzt werden.
2. **Schemaebene**: Ein Schema kann mehrere Würfel enthalten. Für den Großteil der Anwendungsfälle ist jedoch ein Würfel ausreichend. Zusätzlich können die Datentypen der Schemata definiert werden. Die Datentypen können sich je nach physischer Implementierung unterscheiden. Eine MS-SQL Datenbank verwendet z.B. andere Datentypen als eine Oracle Datenbank.
3. **Würfelebene**: Auf der Würfelebene lässt sich eine weitere Unterscheidung zum Modell von [Luján-Mora et al. 2002] erkennen. [Luján-Mora et al. 2002] stellt in der äquivalenten Ebene seines Modells Fakt sowie Dimension als Paket dar. Im GSA-Metamodell wird jedoch nur die Dimension als Paket dargestellt. Fakten werden als Klassen, inklusive der enthaltenen Kenngrößen als Attribute, in dieser Ebene dargestellt. Diese Umstellung begründet sich dadurch, dass in den Kenngrößen der Fakten keine Hierarchie definiert werden muss und somit eine weitere Aufspaltung der Fakten in Klassen nicht notwendig ist. Eine Darstellung als Klasse mit den Kenngrößen als Attributen ist ausreichend.
4. **Dimensionsebene**: Der Aufbau Dimensionsebene besteht zu großen Teilen aus dem von [Akoka et al. 2006] vorgeschlagenen Modell. Die Hierarchie besteht aus Beziehungen zwischen den verschiedenen Aggregationsebenen. Die in einer Aggregationsebene enthaltenen Attribute lassen sich wiederum in identifizierende und nicht identifizierende Attribute unterteilen. In diesem Punkt ergibt sich ein weiterer Unterschied zum Modell von [Luján-Mora et al. 2002], in welchem diese Attribute nicht berücksichtigt werden.

Im Gegensatz zu den anderen Ansätzen muss beim GSA-Metamodell aufgrund vorgegebener Einschränkungen von Eclipse bei der Modellierung auf die Wiederverwendbarkeit von Dimensionen verzichtet werden, da der von Eclipse vorgegebene grafische Editor diese Funktionalität nicht unterstützt.

5.4 Zusammenfassung

Die Notwendigkeit eines Metamodells ergibt sich aus der geforderten Verknüpfung von Data Marts aus verschiedenen Datenquellen. Um die Data Marts vergleichen zu können und darauf aufbauend Mappings zu erstellen müssen einheitliche Regeln und Konzepte festgelegt werden.

Zu Beginn werden die an das Metamodell gestellten Anforderungen beschrieben. Hierzu zählen vor allem Kompatibilität und Übersichtlichkeit des Modells. In weiterer Folge werden bereits bestehende Ansätze zur Erstellung eines Metamodells für DWs beschrieben und verglichen.

Aufbauend auf den Resultaten der Evaluierung der DW Metamodelle wurde schlussendlich ein neues Metamodell, das GSA-Metamodell, erstellt. Für die Erstellung wurden einzelne Elemente der Ansätze der anderen DW Metamodelle herangezogen.

6 Entwicklung des Global Schema Architects

In diesem Kapitel wird zu Beginn das eingesetzte Vorgehensmodell und die damit verbundene Methodik zur Entwicklung des Global Schema Architects beschrieben. Im Anschluss werden die Anforderungen an das Werkzeug im Detail erläutert. Nach einer Übersicht über die angewendeten Technologien wird auf die Architektur des Werkzeugs, sowie die endgültige Implementierung eingegangen.

6.1 Vorgehensmodell

Als Vorgehensmodell wurde das Prototyping-orientierte Prozessmodell nach [Pomberger & Pree 2004 S. 26] angewendet. Durch den Einsatz von Prototypen soll die frühe Klärung von Benutzerbedürfnissen erreicht und Entwicklungsproblemen vermiede werden. Zu diesem Zweck muss stets eine ausführbare Version oder zumindest eine ausführbare Simulation vorhanden sein.

Grundsätzlich lassen sich drei verschiedene Arten von Prototyping unterscheiden:

- **Exploratives Prototyping**: Ziel eines explorativen Prototypen ist die Entwicklung einer möglichst vollständigen Systemspezifikation. Er dient demnach zur Abklärung der Aufgaben und als Grundlage für den Realisierungsweg des Systems.
- **Experimentelles Prototyping**: Zur Abklärung verschiedener technischer Schwierigkeiten und Probleme wird der experimentelle Prototyp verwendet. Er wird in erster Linie zum Testen von neuen Technologien und Konzepten eingesetzt, wodurch Probleme vor der Entwicklung des finalen Produkts erkannt und dadurch vorzeitig die richtigen Maßnahmen gesetzt werden können. Er dient dem System- und Komponentenentwurf.
- **Evolutionäres Prototyping**: Ziel eines evolutionären Prototypens ist eine inkrementelle Systementwicklung, die eine schrittweise Annäherung an die gestellten Benutzeranforderungen erlaubt. Obwohl zwischen Produkt und Prototyp kaum Unterschiede festzustellen sind, werden frühe Versionen aufgrund ihrer geringen Reife als Prototypen bezeichnet.

Für die Erstellung des Global Schema Architects wurde ein evolutionäre Prototyp herangezogen. Das Prozessmodell des Prototypen gliedert sich in Anlehnung das bekannte Wasserfallmodell (siehe [Pomberger & Pree 2004]) in folgende Phasen: *Problemanalyse*, *Systemspezifikation*, *Entwurf*, *Implementierung*, *Systemtest, Betrieb* und *Wartung*. Im Gegensatz zum Wasserfallmodell gibt es jedoch mehrere Entwicklungszyklen, infolgedessen werden die genannten Phasen iterativ mehrmals durchlaufen. Abbildung 6-1stellt den iterativen Ablauf der Entwicklungszyklen grafisch dar.

Ein evolutionärer Prototyp wird solange verbessert bis er die vertraglich spezifizierte Qualität erreicht hat. Diese kann im Zuge des Entwicklungsprozesses in Hinsicht auf neue Anforderungen (z.B. technische Abweichungen) variieren. Im Gegensatz zu den anderen Arten von Prototypen wird dieser niemals verworfen, infolgedessen handelt es sich um einen wiederverwendbaren Prototypen.

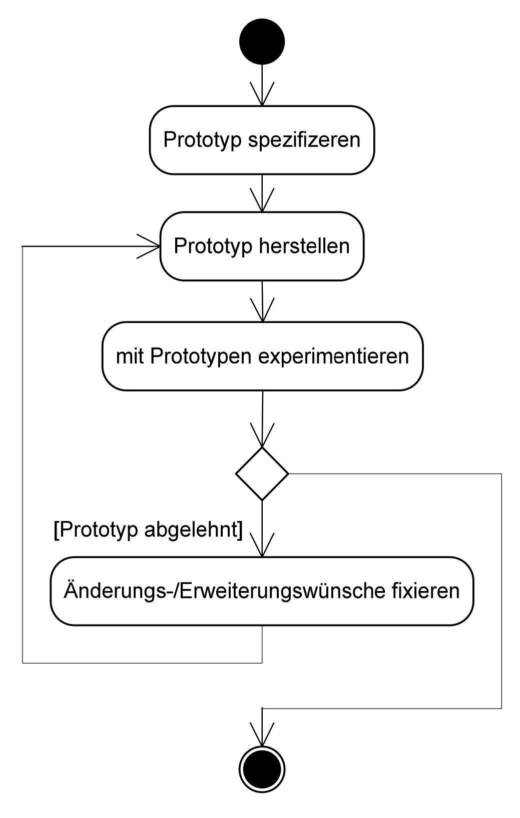


Abbildung 6-1: Prototyp-Aktivitäten (angelehnt an [Pomberger & Pree 2004])

Eine weitere Stärke des evolutionären Prototypings ist die projektbegleitende Qualitätssicherung womit Spezifikations- und Entwurfsfehler früh erkannt und behoben werden können. Zu den Schwierigkeiten zählt u.a., dass die Anforderungen des Kunden mit dem Fortlauf des Pro-

jekts ansteigen könnten, nach dem Prinzip: „*the more they get, the more they want*". [Pomberger & Pree 2004]

6.2 Überblick über die Funktionalität des Werkzeuges

Bevor die einzelnen Anforderungen in Kapitel 6.3 im Detail besprochen werden, gibt dieses Kapitel eine grobe Übersicht über den Umfang des Werkzeuges. Der Global Schema Architect ist Teil eines föderierten DW-Systems und wird durch den SQL-MDi Query Parser sowie den SQL-MDi Query Processor ergänzt. Deren Anwendungsbereiche werden im Folgenden näher beschrieben.

Die grundlegende Funktion des Global Schema Architects ist es die notwendigen Daten für eine Durchführung des Abfrageprozesses (siehe Abbildung 1-1) durch den SQL-MDi Query Parser, sowie durch den SQL-MDi Query Processor vorzubereiten. In Abbildung 1-1 werden die bereitzustellenden Daten grau schattiert hervorgehoben. Infolgedessen ist der Global Schema Architect der Ausführung des Abfrageprozesses vorgelagert.

- Zu den Kernfunktionalitäten des **Global Schema Architects** zählen die Bereitstellung des Global-Schemas, der Import von Data Marts sowie die Ableitung der Daten aus den lokalen Data Marts für das Metadata-Dictionary. Zusätzlich muss der Global Schema Architect eine SQL-MDi Datei bereitstellen, welche die Heterogenität zwischen dem importierten und dem globalen Schema überbrückt.
- Durch den **SQL-MDi Query Parser** wird die vom Global Schema Architect bereitgestellte SQL-MDi Datei in eine Baumstruktur zerlegt um eine weitere Verarbeitung zu ermöglichen. Vor diesem Schritt wird die Struktur durch den SQL-MDi Query Parser validiert [Brunneder 2008].
- Die von SQL-MDi Query Parser bereitgestellte Baumstruktur wird durch den **SQL-MDi Query Processor** zu einem Abfrageplan weiterverarbeitet, welcher aus verschiedenen Teilabfragen für die unterschiedlichen autonomen Data Marts besteht. Basierend auf den verschiedenen Resultaten dieser Abfragen formt der SQL-MDi Query Processor durch ein internes Tabellensystem ein einheitliches Ergebnis. Nähere Details zu diesem Arbeitsschritt können in [Rossgatterer 2008] nachgelesen werden.

6.3 Anforderungen

In den anschließenden Abschnitten werden die Anforderungen an den Global Schema Architect im Detail beschrieben.

6.3.1 Funktionale Anforderungen

Das Werkzeug Global Schema Architect wurde als Projekt am Institut für Data and Knowledge Engineering durchgeführt um die Funktionalität des bereits vorhanden SQL-MDi Query Processors (siehe Abbildung 2-9) zu erweitern.

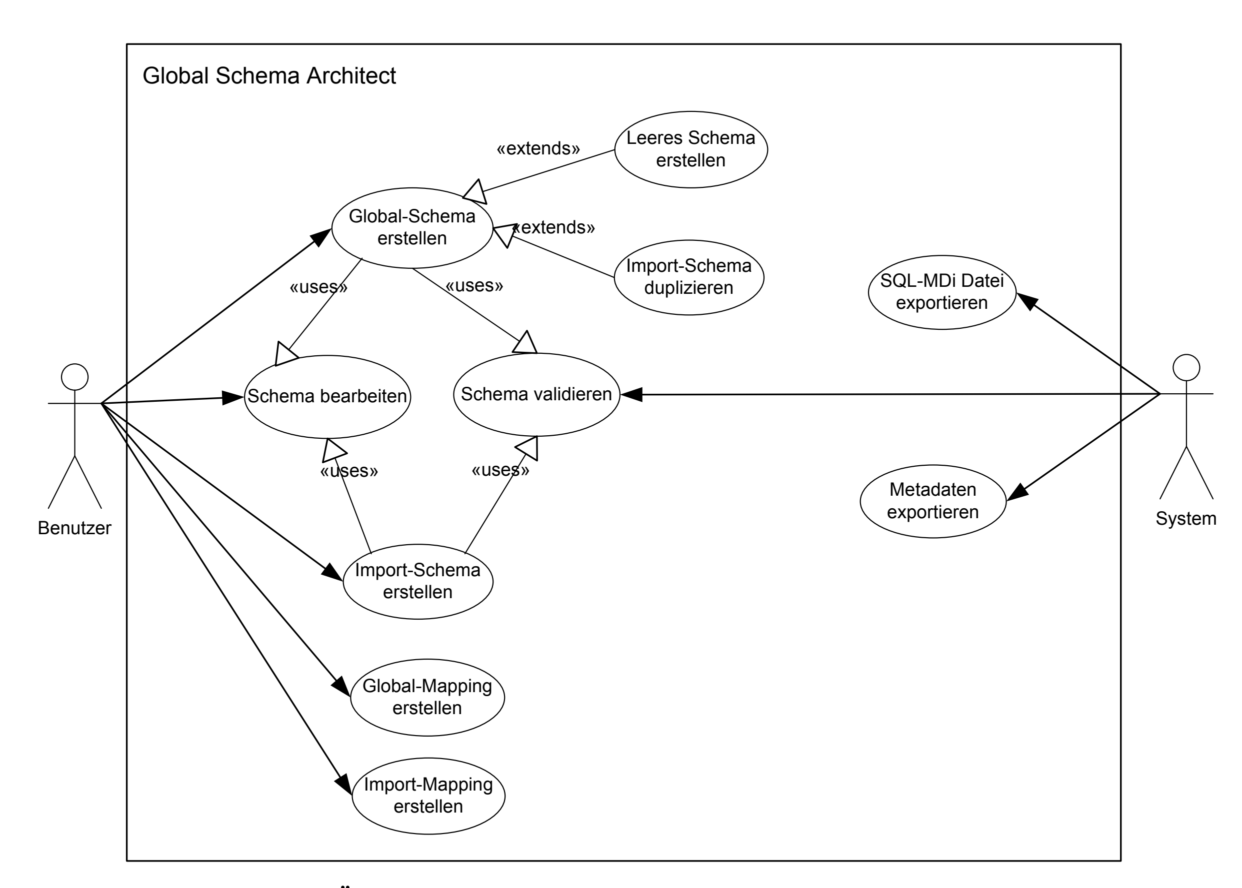


Abbildung 6-2: Überblick über die Anwendungsfälle des Global Schema Architects

Abbildung 6-2 gibt einen Überblick über die im Global Schema Architect geforderten Anwendungsfälle. Die genauen Anforderungen zu den jeweiligen Anwendungsfällen bzw. den Global Schema Architekt werden in den folgenden Kapiteln detailliert beschrieben.

6.3.1.1 Erstellung eines globalen Schemas

Dem Benutzer muss die Möglichkeit gegeben werden, ein globales Schema zu erstellen, welches auf dem in Abschnitt 5.3 beschriebenen GSA-Metamodell basiert. Alle mit dem Metamodell beschreibbaren Konstruktionen müssen durch das globale Schema ausgedrückt werden können. Zu diesem Zweck stellt der Global Schema Architect das Modell grafisch in Form eines UML-Klassendiagramms, zuzüglich der Erweiterungen des zugrundeliegenden Metamodells (siehe Abschnitt 5.3), dar. Zu den zu verwendenden Elementen zählen Würfel, Fakt, Kenngröße, Dimension, Aggregationsstufe und Dimensionsattribut.

Die Erstellung des Global-Schemas erfolgt generell durch den „blank sheet of paper“ Ansatz, was einer Erstellung von Grund auf entspricht. Optional besteht die Möglichkeit ein importiertes Schema in ein globales Schema umzuwandeln. Die Erstellung eines neuen Schemas aus einem bereits importierten Schema beschleunigt die Entwicklungszeit, da aufgrund der vielen Gemeinsamkeiten oftmals die Anpassung eines bestehenden Schemas weniger Zeit in Anspruch nimmt als der Neuaufbau.

Die nachträgliche Bearbeitung des Schemas, das in Form eines Klassendiagramms dargestellt wird, erfolgt weitgehend über „Drag and Drop“-Anweisungen. Darunter versteht man Anweisungen, welche dem Benutzer durch bloßes Ziehen und Klicken, die Erstellung von verschiedenen Elementen ermöglichen. Die Darstellung des grafischen Editors muss sich an der Struktur des grundlegenden Metamodells orientieren, weshalb die Ebenen auch in den grafischen Editor übernommen werden.

Um fehlerhaften Konstruktionen vorzubeugen, stellt der Global Schema Architect die Möglichkeit der Validierung bereit. Ein Validierungsassistent weist gegebenenfalls auf die vorhandenen Warnungen und Fehler hin. Zur Bestimmung von Fehlerquellen werden Regeln und Einschränkungen (sogenannte „Constraints“) definiert, welche Abweichungen vom zugrunde liegenden Metamodell feststellen können.

6.3.1.2 Import von physischen und Ableiten der logischen Schemata

Ziel dieser Anforderung ist es die multidimensionalen Konstrukte eines physischen DW zu identifizieren und daraus die endgültigen Schemata abzuleiten. Diese müssen dem Global

Schema Architect zugrunde liegenden Metamodell entsprechen. Die Anwendung des Metamodells soll den Vergleich mit dem globalen Schema vereinfachen.

Beim Import muss der Benutzer die Zugangsdaten für die gewünschte Datenbank eingeben. Diese Werte müssen sofort validiert werden, wodurch Fehlern beim Zugriff auf die Datenbank entgegen gewirkt wird. Vor dem endgültigen Import muss der Benutzer Anpassungen am Importvorschlag des Systems vornehmen können. Deshalb muss die Definition einer Tabelle (Dimension oder Fakt) anpassbar sein. Zur Erleichterung des Imports wird dem Benutzer ein Assistent zur Verfügung gestellt.

Zusätzlich muss das erstellte Modell auch nach dem Import noch verfeinert und ausgebessert werden können. Dies soll mit dem grafischen Editor, wie bereits in Abschnitt 6.3.1.1 beschrieben, ermöglicht werden.

6.3.1.3 Zuordnung des lokalen Schemas zum globalen Schema (Mapping)

Ziel dieser Anforderung ist es Konflikte zwischen dem Schema eines globalen Data Marts und dem Schema eines lokalen Data Marts zu beseitigen. Die Konflikte zwischen den Data Marts werden mittels sogenannter Mappings überbrückt. Dabei handelt es sich um eine Abfolge von Operatoren der Fact Algebra bzw. der Dimension Algebra (siehe [Berger & Schrefl 2008]), welche im Global Schema Architect von der multidimensionalen Abfragesprache SQL-MDi ausgedrückt werden.

Zum Beseitigen der Konflikte zwischen lokalen und globalen Schema muss dem Benutzer ein Assistent bereitgestellt werden, der die Erstellung der Mappings unterstützt. Der Mapping-Assistent zeigt für jedes Mapping-Konstrukt „kontextsensitiv" nur jeweils die anwendbaren Operatoren an. Aus den verschiedenen Einstellungen im Mapping-Assistent erstellt der Global Schema Architect automatisch die entsprechenden SQL-MDi Ausdrücke.

Die Erstellung des Mappings erfolgt inkrementell. So werden die Einstellungen im Mapping-Assistenten Schritt für Schritt durchgeführt. Eine „all-at-once"-Lösung ist dezidiert nicht gefordert.

Es werden zwei Arten von Mappings unterschieden:

- Mit dem **Import-Mapping** können Konflikte auf Schemaebene aufgelöst werden. Zum Auflösen der Konflikte werden unäre Operatoren eingesetzt. Das Import-Mapping verwendet das Import-Schema als Basis, demnach werden die Heterogenitäten aus Sicht des Import-Schemas zum Global-Schema beseitigt. Die Elemente des Import-Schemas bilden somit die Komponenten des Mapping-Assistenten. Bei der Erstellung eines Mappings kann demzufolge von den Elementen eines multidimensionalen Modells (Würfel, Dimension, Fakt, Aggregationsebene und Kenngröße) ausgegangen werden. Beispielsweise werden von einer Dimension im Import-Schema ausgehend Diskrepanzen zur entsprechenden Dimension im Global-Schema beseitigt. Die Anordnung der Elemente muss der von SQL-MDi geforderten Reihenfolge entsprechen. Für jedes importierte Schema muss eine eigene Datei angelegt werden.
- Das **Global-Mapping** löst Konflikte durch n-äre Operatoren auf Instanzebene auf. Die Erstellung des Global-Mappings ist der Erstellung der Import-Mappings im Entwicklungsprozess nachgelagert. Demnach werden dort die Konflikte zwischen den einzelnen Instanzen der Import-Schemas behoben. Aufgrund der ganzheitlichen Sicht des Global-Mappings ist der Einsatz einer einzigen Datei ausreichend. Das Global-Mapping basiert nicht auf einem einzigen Schema, sondern auf einer Vielzahl von Schemata, weshalb die Notwendigkeit zur dynamischen Erstellung von Mapping-Komponenten gegeben ist. Zu den Mapping-Komponenten zählen entsprechend der SQL-MDi Syntax die Anweisungen *Merge Dimensions* und *Merge Cubes*.

Abbildung 6-3 beschreibt den Import- und den Mapping-Prozess. Aus den vorgegebenen Datenquellen (engl. Data Sources) werden die lokalen Schemata aus dem Data Mart ausgelesen und als Import-Schema abgebildet. Daraufhin wird jedes Import-Schema für sich durch Import-Mappings auf das globale Schema abgebildet. Darüberhinaus wird ein einzelnes globales Mapping zum Beseitigen der Konflikte der Schemainstanzen benötigt.

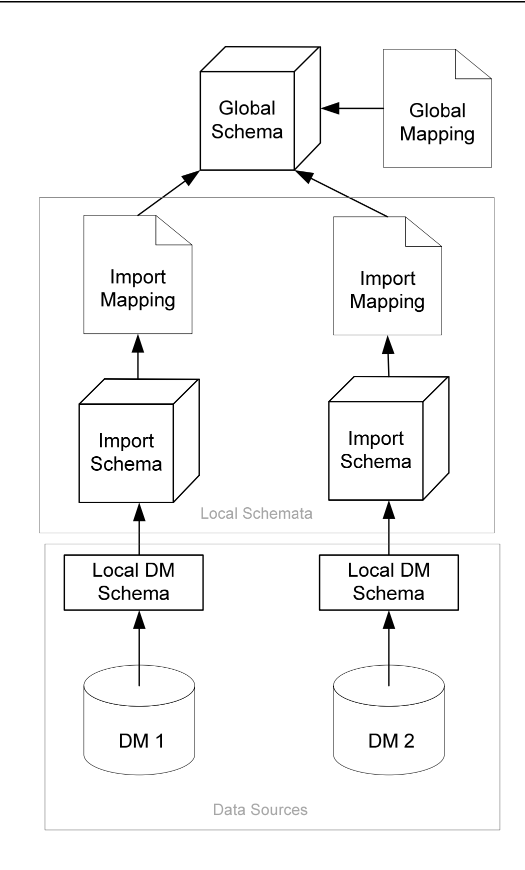


Abbildung 6-3: Beschreibung von Import und Mapping

6.3.1.4 Speicherung der Metadaten

Abbildung 1-1 veranschaulicht die zentrale Verwendung des Metadata-Dictionaries als Schnittstelle zwischen dem Global Schema Architect und dem SQL-MDi Query Parser. Die Struktur der Metadaten wird vom SQL-MDi Query Parser vorgegeben und muss vom Global Schema Architect dementsprechend umgesetzt werden. Die Speicherung der Metadaten erfolgt in einer eigenen Datenbank, dem Metadaten-Repository.

Zur Durchführung des Exports verlangt der Export-Assistent vom Benutzer lediglich die Eingabe der Verbindungsdaten zur Datenbank. In weiterer Folge muss der Export-Assistent die Modelle aus den verschiedenen Schemata auslesen und auswerten um eine korrekte Speicherung der Metadaten zu berücksichtigen. Es dürfen z.B. Dimensionen welche mehrmals verwendet wurden nur einmal im Metadata-Dictionary verzeichnet werden. Schlussendlich werden die notwendigen Einfüge-Operationen an die Datenbank übermittelt. Zusätzlich zu den Metadaten des Metadata-Dictionaries müssen die Metadaten zu den Dimensionen des globalen Schemas in das Dimension-Repository gespeichert werden.

6.3.1.5 Bereitstellung einer Schnittstelle für Abfragen

Neben den exportierten Metadaten wird eine SQL-MDi Datei als Schnittstelle zu anderen Anwendungen bereitgestellt und zum Überbrücken von Heterogenität eingesetzt.

Die endgültige SQL-MDi Datei muss ein Zusammenschluss der in den unterschiedlichen Mapping-Dateien (Import- und Global-Mapping) erzeugten SQL-MDi Fragmente sein, wobei eine der Syntax entsprechende Anordnung der Elemente essentiell ist. In der SQL-MDi Datei müssen die Ausdrücke der Import-Mappings vor den Global-Mappings angeführt werden. Der beschriebene Prozess muss von einem entsprechenden Assistenten unterstützt werden. Auf Seiten des Benutzers muss als einzige Interaktion ein Name für die zu erstellende Datei vergeben werden.

Während des Exports der SQL-MDi Datei überprüft das System die Datei auf seine syntaktische Korrektheit. Sollten Fehler auftreten müssen dem Benutzer die entsprechenden Fehlermeldungen angezeigt werden. Die Validierung, sowie die möglichen Fehlermeldungen werden dem von [Brunneder 2008] entwickelten SQL-MDi Query Parser entnommen.

6.3.2 Nicht funktionale Anforderungen

Neben der Umsetzung der funktionalen Anforderungen sind auch Anforderungen an die Beschaffenheit und die Rahmenbedingungen zu klären. Nicht funktionale Anforderungen sind ebenso wie die funktionalen Anforderungen für den Erfolg eines Projekts entscheidend und müssen daher auch klar definiert werden.

6.3.2.1 Plattform und Programmiersprache

Für die Umsetzung bzw. spätere Integration des Werkzeuges als Plug-in wird die weit verbreitete Rich-Client-Plattform Eclipse (siehe Kapitel 4) eingesetzt werden.

Bei der Entwicklung des Plug-ins müssen die von Eclipse vorgegeben Standards und Frameworks verwendet werden. In diesem Projekt werden im Speziellen der Einsatz des Eclipse UML2-Frameworks, des Eclipse Modeling Frameworks (EMF) und des Graphical Editor Frameworks (GEF) gefordert [Eclipse Tools 2008]. Eine genaue Beschreibung der Eclipse Plattform als auch der dazugehörigen Frameworks findet sich in Kapitel 4.

6.3.2.2 Verständlichkeit und Standards

Der Einsatz von Standards erhöht die Verständlichkeit und gibt dem Benutzer das Gefühl mit bereits bekannten Technologien zu arbeiten. In diesem Projekt kommen CWM (siehe Abschnitt 3.6) als Grundlage des Metamodells sowie UML (siehe Abschnitt 3.5) als Modellierungssprache zum Einsatz. Speziell UML kann als weit verbreitetes, und von der OMG empfohlenes, Werkzeug zur Modellierung einen hohen Wiedererkennungswert bei den Benutzern erwirken [Gruhn et al. 2006 S. 282].

6.3.2.3 Benutzerfreundlichkeit

Es wird ausdrücklich ein Schwerpunkt auf die Bedienung des Werkzeuges gelegt. Durch den Einsatz von Eclipse als Anwendungsumgebung für das Werkzeug soll dem Benutzer das Gefühl vermittelt werden, sich in einem vertrauten Umfeld zu befinden. Bei der Entwicklung des GUI muss hoher Wert auf Usability bzw. Selbsterklärungsfähigkeit gelegt werden. Die verschiedenen Elemente des Plug-ins sollen für einen erfahrenen Eclipse-Benutzer intuitiv anwendbar sein.

6.3.2.4 Prototypischer Ansatz

Die Umsetzung des Werkzeuges muss nach dem prototypischen Ansatz durchgeführt werden (siehe Abschnitt 6.1). Mithilfe des Prototypens sollen die geforderten funktionalen Anforderungen umgesetzt und evaluiert werden können. Für die Umsetzung wird von relationalen Star-Schemata ausgegangen. In Bezug auf die Datenbanken wird demnach keine vollständige Systemunabhängigkeit gewährleistet, da die Realisierung des Prototyps sich auf eine exemplarische Realisierung für eine Auswahl von Datenbanksystemen beschränkt hat. Der Einsatz in einer Produktivumgebung ist explizit nicht Ziel dieser Diplomarbeit.

6.4 Technologien und Standards

In diesem Kapitel wird auf für die Implementierung des Werkzeuges notwendige Standards- und Technologien eingegangen.

6.4.1 Eclipse 3.4

In den Anforderungen wurde eine Umsetzung als Eclipse Plug-in gefordert, demzufolge ist Eclipse als Entwicklungsumgebung unumgänglich. Eine genaue Beschreibung von Eclipse und dessen Frameworks und Technologien findet sich in Kapitel 4.

Zur Entwicklung wurde Eclipse 3.4 verwendet. Von den von Eclipse vorgegebenen Frameworks werden als Mindestanforderungen die Versionen EMF 2.4 und UML2 2.2 vorgegeben.

6.4.2 Java 1.6

Die Entwicklung als Eclipse-Plug-in setzt Java als Programmiersprache voraus. Als Java Version wurde die zum Zeitpunkt dieser Arbeit aktuelle Version 1.6 herangezogen.

Java eignet sich aufgrund der objektorientierten Struktur für Anwendung einer „Model Driven Architecture“. Modelle können wie objektorientierte Sprachen in Objekte, Pakte und Klassen unterteilt werden. Aus diesem Grund sind Konstrukte der MDA relativ einfach in objektorientierte Sprachen abzubilden. [Gruhn et al. 2006 S. 13ff]

Java ermöglicht eine plattformunabhängige Entwicklung. Die Unabhängigkeit ergibt sich aufgrund der Java Virtual Machine (JVM), welche als Schnittstelle zwischen Maschine und Betriebssystem auftritt. Durch diese Zwischenebene können Java-Anwendungen ohne jegliche Adaptionen auf allen unterstützten Plattformen ausgeführt werden. Damit wird die Vorgabe der Plattformunabhängigkeit erfüllt [Ullenboom 2007].

6.4.3 JDBC

Mittels JDBC können Datenbankinteraktionen zwischen Java-Anwendungen und einer Vielzahl von verschiedenen Datenbanksystemen erstellt werden. Die Anwendung von Java bietet dem Benutzer die Möglichkeit einen einmal geschriebenen Code auf mehrere Datenbanksysteme anzuwenden, wodurch der Code wiederverwendbar und verständlicher wird [JDBC 2009].

6.4.4 xPath

Die Speicherung von den Einstellungen des Global Schema Architect erfolgt durch XML-Ausdrücke. Zum Auslesen von Information aus den XML-Ausdrücken wird die Technologie xPath eingesetzt (siehe Abschnitt 7.2.2).

Die Hauptaufgabe von xPath ist das Referenzieren bzw. das Auswählen von XML-Elementen in einem XML-Dokument. Zusätzlich gibt es die Möglichkeit einfache Manipulationen an den XML-Elementen durchzuführen. Die Bezeichnung von xPath ergibt sich aus der pfadartigen Darstellung der Syntax, um durch die verschiedenen XML-Strukturen zu navigieren [Clark & Derose 1999].

6.4.5 The Standard Widget Tookit (SWT)

Eclipse forciert die Verwendung von SWT. Durch SWT wird das „*look and feel*" einer Anwendung an die grafischen Vorgaben der Systeme angepasst. Diese Technologie wird implizit im kompletten Projekt verwendet. Speziell bei der Erstellung der Eclipse Forms (siehe Abschnitt 7.2) wird diese Technologie zum Einsatz kommen [SWT 2009].

6.4.6 MS SQL Server 2005 / Oracle 10gR2

Für die Verwendung der Datenbanksysteme wurde aufgrund der geforderten Kompatibilität zum SQL Query Parser der MS SQL Server 2005 und die Oracle 10gR2 Datenbank ausgewählt. Zusätzlich zählen diese Datenbanksystem zu gebräuchlichen ROLAP-Plattformen. Das Dimension-Repository wurde auf Oracle implementiert um standardmäßige OLAP Abfragen erstellen zu können. Das Metadaten-Repository und das Komponentensystem wurden auf dem MS SQL Server umgesetzt [Rossgatterer 2008].

6.5 Architektur des Systems

Dieses Kapitel erläutert die Systemarchitektur des Global Schema Architects. Nach einer Übersicht des ganzen Systems wird auf die einzelnen Komponenten eingegangen bevor deren Implementierung in Abschnitt 7 näher erläutert wird.

Abbildung 6-4 stellt die Systemarchitektur des Werkzeuges dar. Die Darstellung des Systems erfolgt zur Verbesserung der Verständlichkeit auf einer hohen Abstraktionsebene.

Der Global Schema Architekt besteht aus acht verschiedenen Komponenten: *Metamodell*, *Import-Schema*, *Global-Schema*, *Grafischer Editor*, *Import-Mapping*, *Global-Mapping*, *Export Metadata Wizard* und *Export SQL-MDi Wizard*. Der enge Zusammenhang unter den Komponenten wird in Abbildung 6-4 veranschaulicht und manifestiert sich in der zentralen Anwendung des Metamodells. Das Metamodell referenziert die Modellklassen des *Import-Schemas* und des *Global-Schemas*. Die Modelle sind grundlegender Bestandteil des jeweiligen Schemas und werden durch den Import-Assistenten (*Import-Schema*) bzw. durch den Editor zur Erstellung des Schemas (*Global-Schema*) ergänzt. Zur Bearbeitung von *Import-* und *Global-Schema* wird ein *grafischer Editor* herangezogen, welcher auch auf dem *Meta-Modell* basiert.

Die Mapping-Komponenten *Import-Mapping* und *Global-Mapping* greifen auf die entsprechenden Modelle (Import- und Global-Model) zu, wobei der *Global Mapping Editor* nicht auf das *Import-Model* sondern auf das *Import-Mapping* zugreift. Eine detaillierte Beschreibung dieser Problematik findet sich in Abschnitt 6.5.2.

Zu den Export-Komponenten zählen *Export Metadata Wizard* und *Export SQL-MDi Wizard*. Beide Komponenten sind völlig unabhängig voneinander an das System gekoppelt. Für den SQL-MDi Export werden die zuvor in den Mappings erzeugten SQL-MDi Code Fragmente als Grundlage herangezogen. Der Export der Metadaten basiert auf dem *Import-* sowie *Global-Model*.

Um eine bessere Strukturierung zu ermöglichen, ist eine Gruppierung der Komponenten aufgrund der Aufgabenbereiche in drei Hauptgruppen sinnvoll:

- **Schemabezogene Komponenten**: Zu den schemabezogenen Komponenten zählen alle Komponenten welche sich mit dem Import sowie dem Editieren der einzelnen Modelle beschäftigen. Dazu gehören das *Metamodell*, das *Import-Schema*, das *Global-Schema* und der *grafische Editor*.
- **Mapping-Komponenten:** Als Mapping-Komponenten werden alle Komponenten angesehen, welche mit der Erstellung der Mappings beschäftigt sind (*Import-Mapping* und *Global-Mapping*).

- **Export-Komponenten:** Aufgabe der Export-Komponenten ist es die Schnittstellen zu anderen Programmen, wie z.B. dem SQL-MDi Query Parser bereit zu stellen. Hierzu zählen der *Export Metadata Wizard* und der *Export SQL-MDi Wizard.*

Die Einteilung der Hauptgruppen wird auch in der Systemimplementierung (siehe Kapitel 7) eingesetzt.

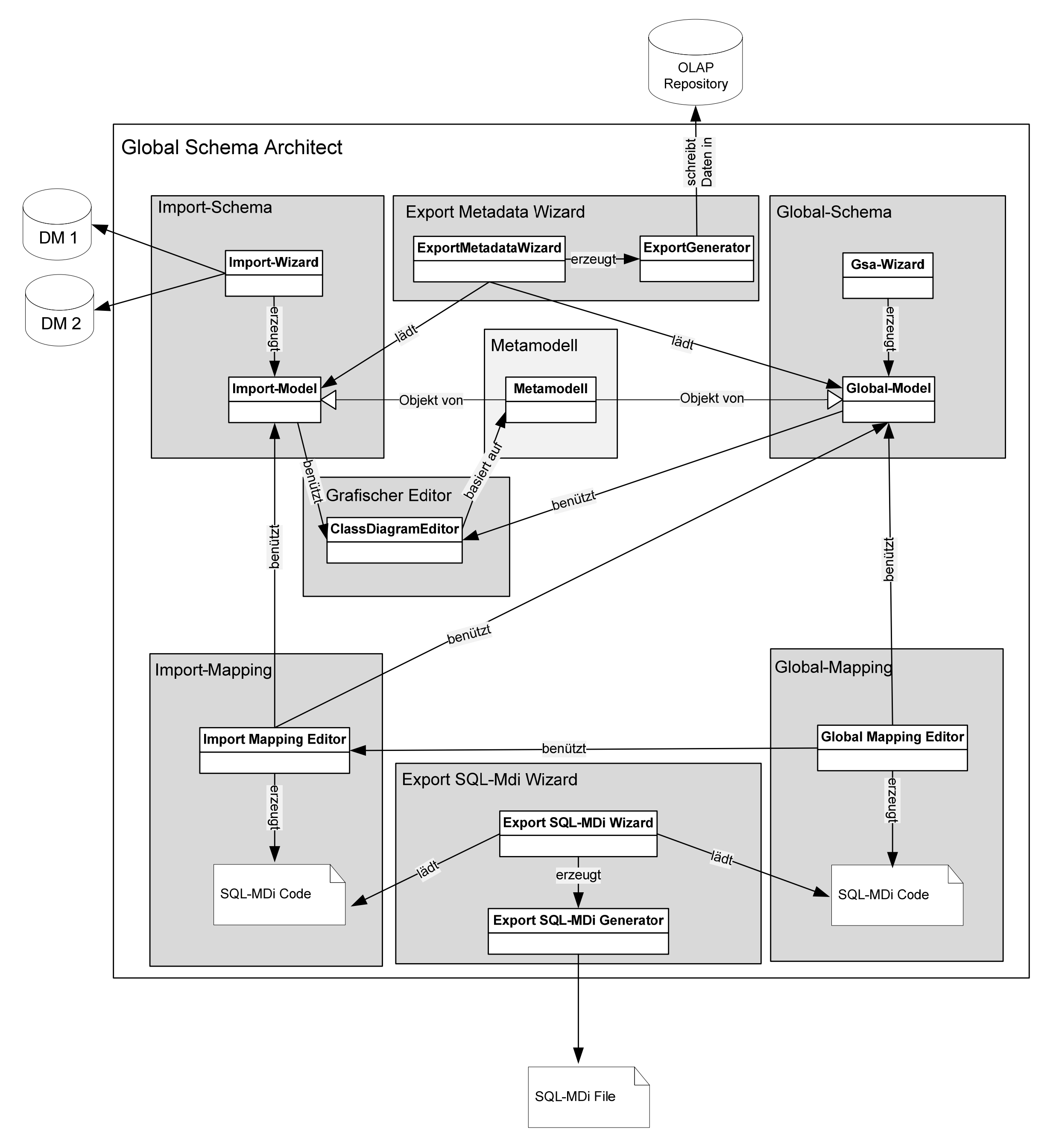


Abbildung 6-4: Systemarchitektur des Global-Schema Architects

6.5.1 Schemabezogene Komponenten

Der Zusammenhang dieser Komponenten ergibt sich aus der direkten Interaktion mit dem Metamodell bzw. aus der Tatsache, dass das Metamodell selbst die Komponente ist. Zum einen müssen Modelle auf Basis des Metamodells erstellt und importiert werden, zum anderen muss eine nachträgliche Bearbeitung möglich sein.

Komponente Metamodell

Das Metamodell ist die grundlegende Komponente des Werkzeuges, da es einerseits den Aufbau der Schemata angibt und andererseits den Grundstein für die Kompatibilität zur externen Schnittstelle setzt. Die Erstellung und der Aufbau des Metamodells wurde bereits in Abschnitt 5.3 ausführlich beschrieben. In Abschnitt 7.1 wird im Detail auf die Implementierung eingegangen.

In Abbildung 6-4 wird die zentrale Position des Metamodells verdeutlicht. Als Grundlage der Modelle des *Import-Schemas* und *Global-Schemas* ist es zumindest indirekt an allen Programmkomponenten beteiligt.

Komponente Import-Schema

Diese Komponente hat primär die Aufgabe vorhandene Schemata aus der Datenbank zu importieren und aufgrund dieser dem Benutzer einen Schemavorschlag bereitzustellen, aus welchem in weiterer Folge das *Import-Model* erzeugt wird. Das *Import-Model* stellt die Schnittstelle zu den anderen Komponenten der Systemarchitektur dar.

Abbildung 6-5 illustriert den Algorithmus des Importprozesses in Form eines UML-Aktivitätsdiagrammes. Bei erfolgreicher Verbindung zur Datenbank werden die Metadaten aus der entsprechenden Datenbank ausgelesen. Nach dem Laden der Metadaten wird ein neues UML-bzw. *Import-Model* erstellt. Im Anschluss werden die Tabellen sowie Spalten der Datenbank traversiert. Solange weitere Tabellen gefunden werden, wird die Tabelle auf exportierte Fremdschlüssel überprüft. Exportierte Fremdschlüssel sind Verweise von anderen Tabellen auf den Primärschlüssel dieser Tabelle. Sollte dies der Fall sein, wird dem UML-Modell eine neue Dimension hinzugefügt. Der Import der Fakten verhält sich analog zur Erstellung der Dimensionen, mit der Abweichung, dass importierte Fremdschlüssel vorhanden sein müssen. Nachdem keine weitere Tabelle mehr gefunden wird, kann die UML-Datei ge-

speichert und in weiterer Folge die für die Erstellung der Fakten und Dimensionen notwendigen Stereotypen an die zuvor erstellten Elemente angebracht werden.

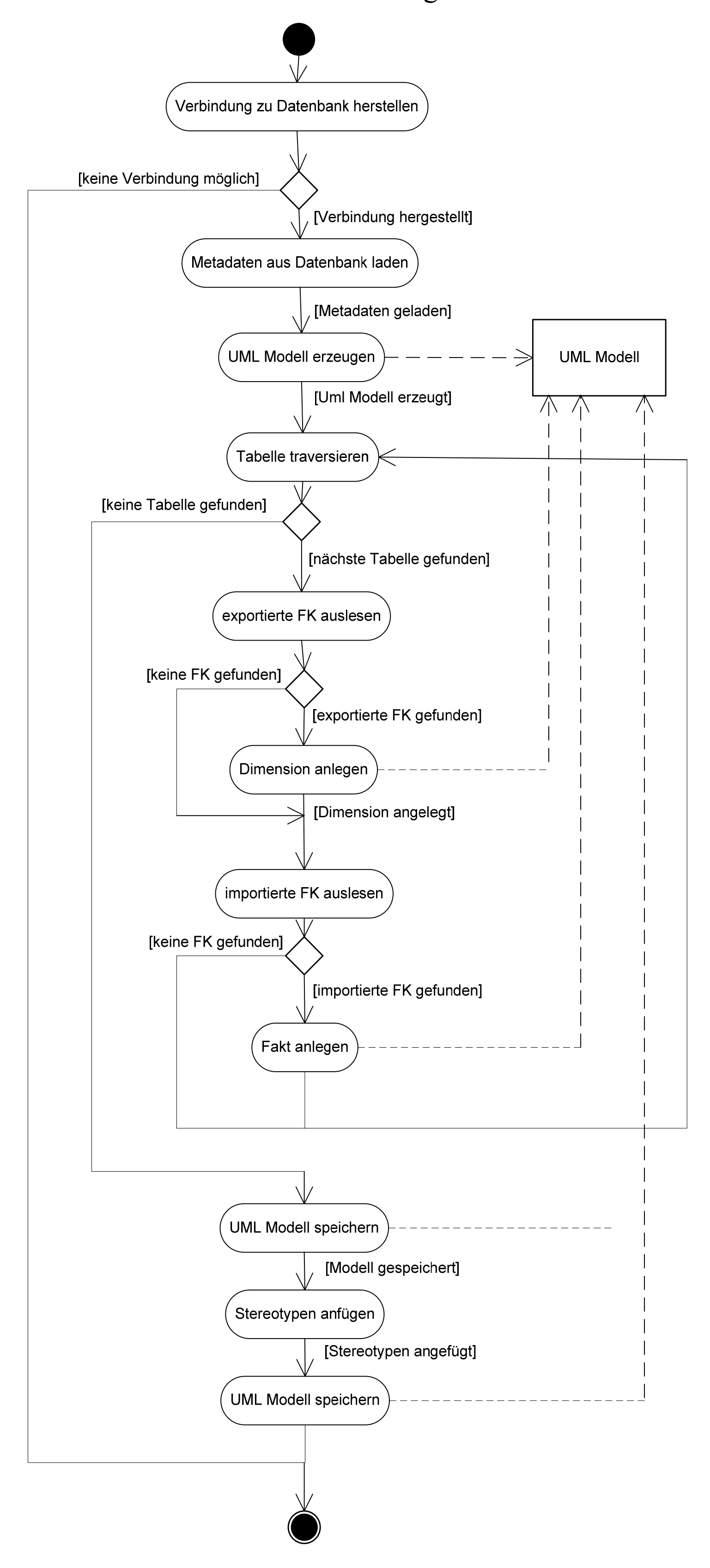


Abbildung 6-5: Funktionalität zum Import von DW

Im Zuge der Erstellung des *Import-Schemas* werden neben dem multidimensionalen Schema zusätzlich die Verbindungsdaten zur Datenbank als Annotationen im jeweiligen *Import-Schema* gespeichert. Zur Speicherung des Passworts wird ein Verschlüsselungsalgorithmus (Klasse `Encryption`) verwendet um die Sicherheit des Werkzeuges gewährleisten zu können. Die Verbindungsdaten werden auch für die Ausführung des Exports der Metadaten benötigt.

Komponente Global-Schema

Das *Global-Schema* verhält sich ähnlich wie das *Import-Schema* mit dem Unterschied, dass die Daten nicht importiert, sondern direkt eingegeben werden. Zur Erstellung des globalen Schemas kann zwischen zwei Assistenten (engl. Wizard) gewählt werden. Zum einen der *Global Schema Architect File* Assistent welcher ein leeres *Global-Model* erzeugt, und zum anderen der *Global Schema From Import Schema* Assistent welcher aus einem *Importmodel* ein neues *Global-Model* mit übereinstimmendem Inhalt erstellt.

Analog zum *Import-Schema* müssen auch beim *Global-Schema* Verbindungsdaten zu einer Datenbank eingegeben werden. Im Gegensatz zum *Import-Schema* werden diese jedoch nicht zum Importieren von Daten, sondern zum Exportieren (siehe Abschnitt 6.5.3) der Dimensionen des globalen Schemas in das Dimension-Repository bereitgestellt.

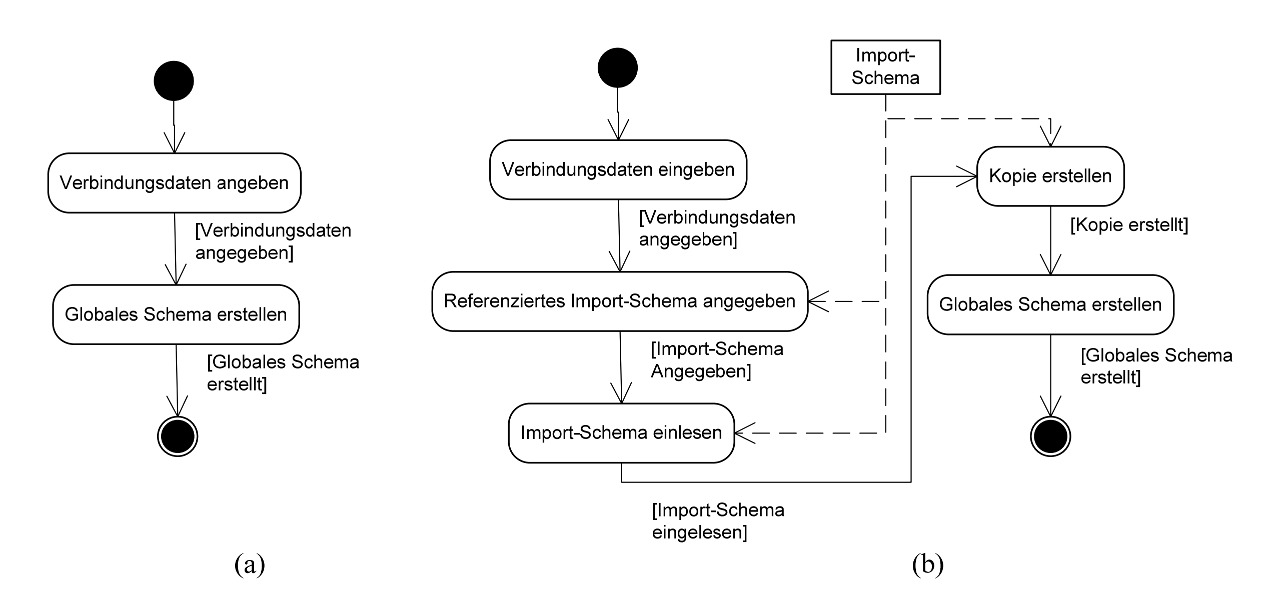


Abbildung 6-6: Aktivitätsdiagramm Global-Schema

Abbildung 6-6 veranschaulicht die verschiedenen Varianten zur Erstellung eines Global-Schemas mithilfe eines Aktivitätsdiagrammes. Abbildung 6-6 (a) illustriert den Prozess zur Erstellung eines leeres Global-Schemas. In Abbildung 6-6 (b) wird der Prozess der Kopie eines Import-Schemas dargestellt. Die Abweichungen ergeben sich daraus, dass bei (b) ein Import-Schema eingelesen und kopiert werden muss.

Komponente Grafischer Editor

Der grafische Editor wird zur Bearbeitung von Modellen benützt. Zur Repräsentation wird das geforderte UML-Klassendiagramm eingesetzt. Zur Umsetzung des Editors werden die von Eclipse bereitgestellten UML2Tools (siehe Abschnitt 4.4.2) verwendet. Die Anbindung an den Global Schema Architect erfolgt über die Standardversion der UML2Tools, es sind lediglich marginale Erweiterungen vorzunehmen.

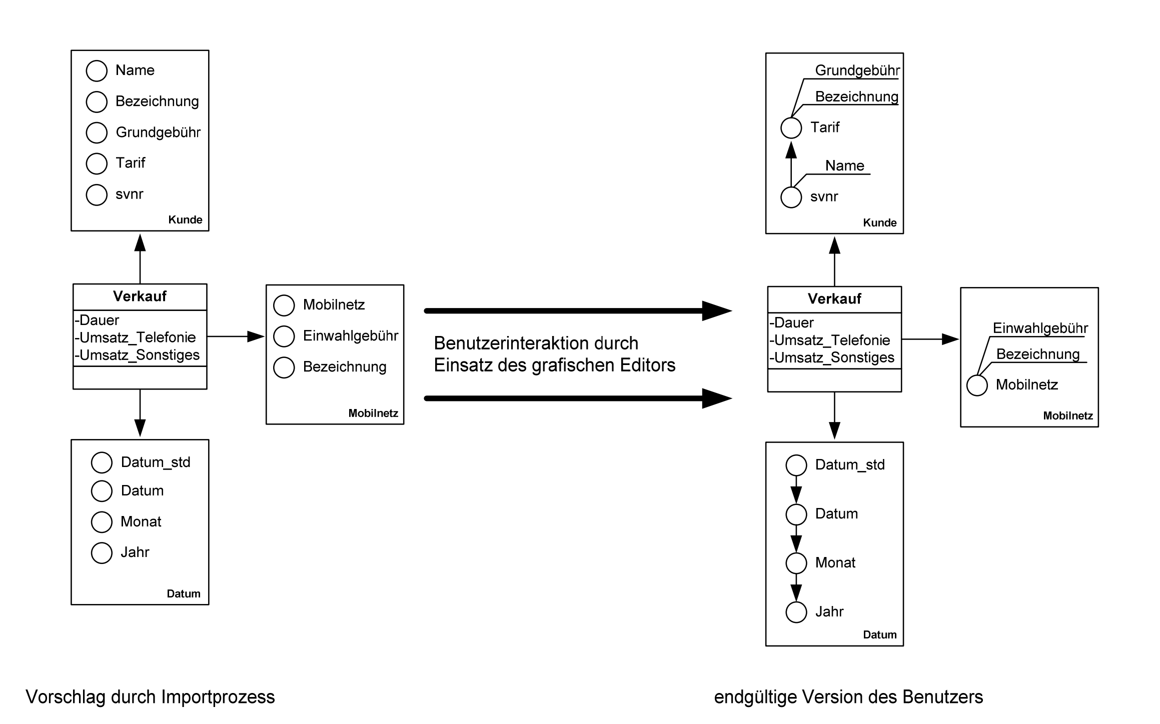


Abbildung 6-7: Benutzerinteraktion durch grafischen Editor

Abbildung 6-7 zeigt einen Anwendungsfall für den Einsatz des grafischen Editors. Die linke Seite der Abbildung beschreibt den Zustand des Modells nach dem Import. Die rechte Seite zeigt das fertige Modell, das der Benutzer durch Anwendung des grafischen Editors erstellt hat.

6.5.2 Mapping-Komponenten

Durch Mappings werden Heterogenitäten zwischen den verschiedenen Modellen beseitigt. Für jedes *Import-Schema* gibt es ein Mapping das es auf die Struktur des *Global-Schemas* anpasst. Es wird somit eine Vielzahl von *Import-Mappings* eingesetzt. Die Anzahl der *Import-Mappings* muss stets der Anzahl der *Import-Schemas* entsprechen. Zusätzlich wird ein *Global-Mapping* eingesetzt, welches den anderen Mappings übergeordnet ist und Konflikte zwischen dem *Global- Schema* und allen *Import-Mappings* behebt.

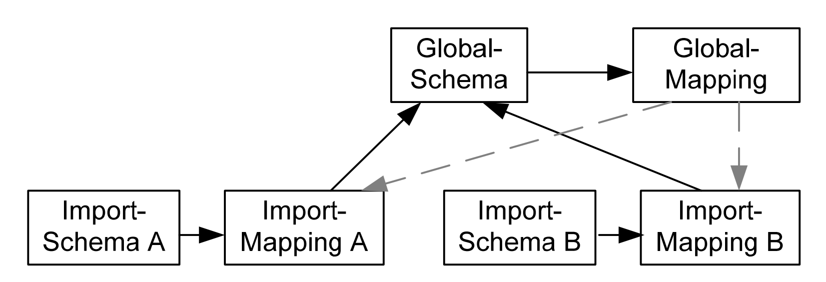


Abbildung 6-8: Aufteilung der Mappings

Die Erstellung der SQL-MDi Datei zur Überbrückung der Heterogenitäten zwischen den einzelnen Import-Schemas erfolgt auf inkrementell. Dieser Prozess wird in Abbildung 6-9 dargestellt. Der Benutzer fügt sukzessive SQL-MDi Ausdrücke der SQL-MDi Datei hinzu. In diesem Zusammenhang hat er die Möglichkeit sich vom System einen Vorschlag für den entsprechenden SQL-MDi Ausdruck generieren zu lassen. Das Erstellen der Ausdrücke gliedert sich nach den Elementen des jeweiligen Schemas.

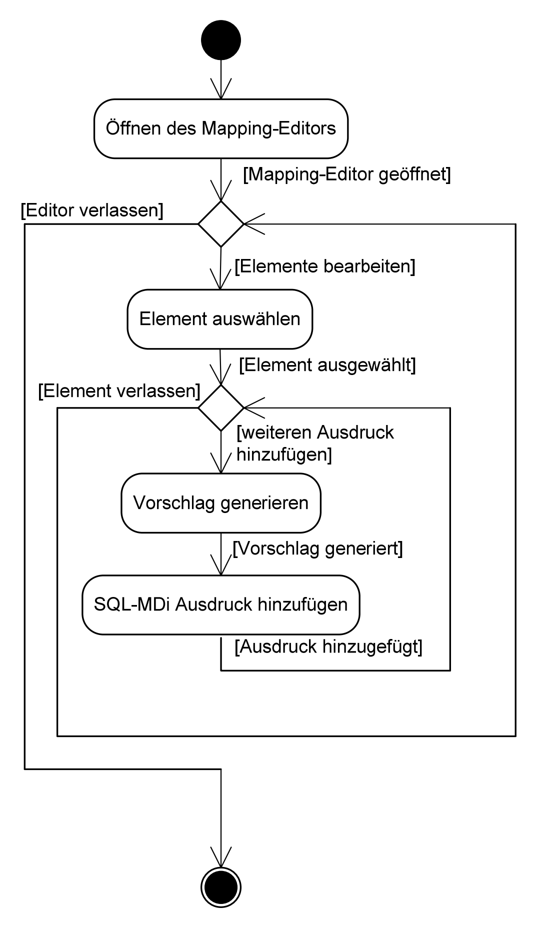


Abbildung 6-9: Aktivitätsdiagramm Hinzufügen von SQL-MDi Ausdrücken

Import-Mapping

Mithilfe des *Import-Mappings* werden die Heterogenitäten zwischen *Global-Schema* und *Import-Schema* überbrückt. Für die Erstellung eines *Import-Mappings* wird sowohl das *Global-Schema* als auch das *Import-Schema* herangezogen. Nach dem Import hat der Benutzer die Möglichkeit die einzelnen Heterogenitäten inkrementell zu beseitigen.

Für das *Import-Mapping* ist die Anzahl der Elemente fest vorgegeben. So werden alle Elemente des *Import-Schemas* (wie Würfel, Dimension, Dimensionsebene, Fakt und Kenngröße) in das Mapping geladen und daraufhin dem *Global-Schema* zugeordnet.

Um das nachträgliche Bearbeiten von Mappings zu ermöglichen, müssen diese in einem entsprechenden Format abgespeichert werden. In dieser Arbeit wurde dazu XML eingesetzt. Weiters müssen auf Basis des XML-Codes SQL-MDi Fragmente erzeugt werden können, welche als Grundlage für Export und Vorschau verwendet werden. Abbildung 6-10 veran-

schaulicht diesen Prozess auf abstrakter Ebene. Für jedes zu speichernde Element sind Abweichungen vom dargestellten Prozess möglich.

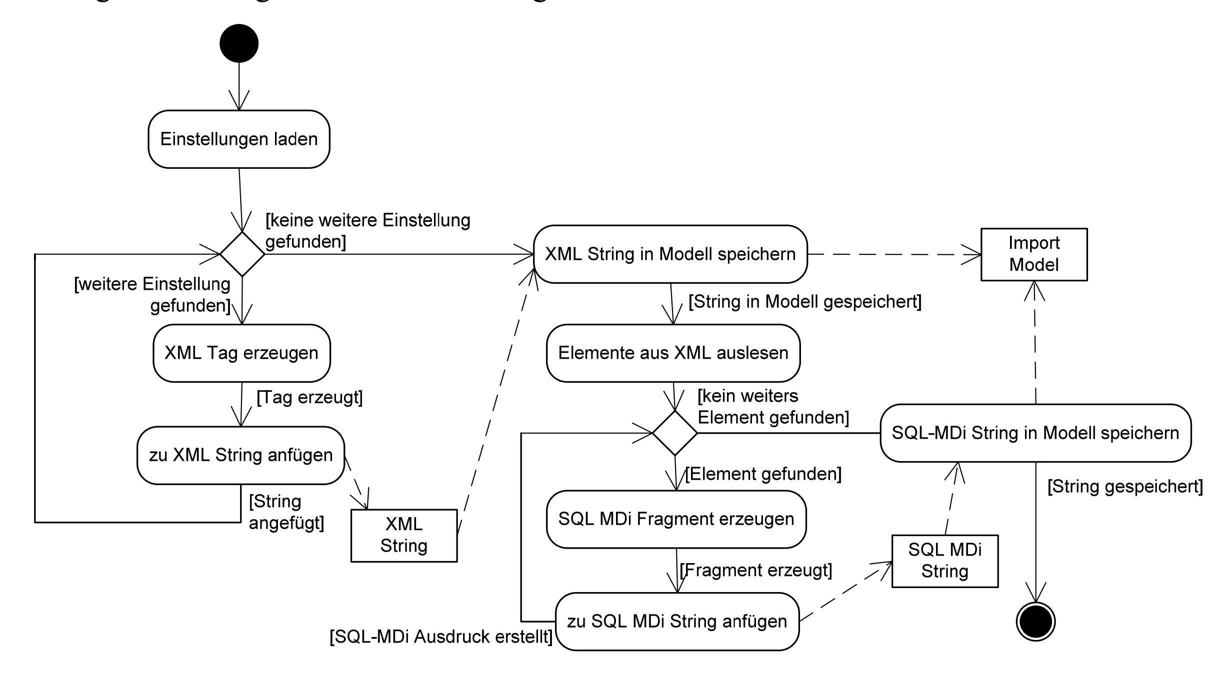


Abbildung 6-10: Funktionalität Speicherprozess Import-Mapping

In Abbildung 6-10 werden zu Beginn die Einstellungen aus der grafischen Oberfläche geladen. In Anschluss werden die XML-Ausdrücke zu den jeweiligen Einstellungen gebildet und im Modell gespeichert. Auf Basis der XML-Ausdrücke werden daraufhin die entsprechenden SQL-MDi Ausdrücke erzeugt und wiederum in das Modell gespeichert.

Global-Mapping

Das *Global-Mapping* hat die Aufgabe die *Import-Mappings* und das *Global-Schema* zusammenzuführen. Pro Projekt gibt es ein *Global-Mapping*, mit dem Konflikte zwischen den einzelnen Instanzen der Import-Schemata sowie des Global-Schemas behoben werden können.

Die Grundfunktionalität des *Global-Mapping* Editors ist in vielerlei Hinsicht mit dem des *Import-Mappings* zu vergleichen. Der größte Unterschied ergibt sich aus der dynamischen Erzeugung der Elemente. Die zu verwendenden Elemente sind entweder *MERGE CUBES* oder *MERGE DIMENSIONS* Anweisungen. So kann z.B. immer wieder eine *MERGE DIMENSIONS* Anweisung hinzugefügt werden.

Ein weiterer Unterschied besteht darin, dass das *Global-Mapping* keine direkte Beziehung zu dem *Import-Schema* aufweist. Eine direkte Beziehung ist nicht möglich, weil das *Global-Schema* die bereits durch das *Import-Mapping* bereitgestellten Konfliktlösungen berücksichtigen muss. So referenziert das *Global-Mapping* das *Global-Schema* sowie alle *Import-Mappings* des Projekts. Die gespeicherten Daten der *Import-Mappings* werden bei jedem Laden des *Global-Mapping* Editors dynamisch neu übernommen. Dies ermöglicht dem Benutzer eine nachträgliche Bearbeitung der *Import-Mappings* ohne das *Global-Mapping* erneut erstellen zu müssen.

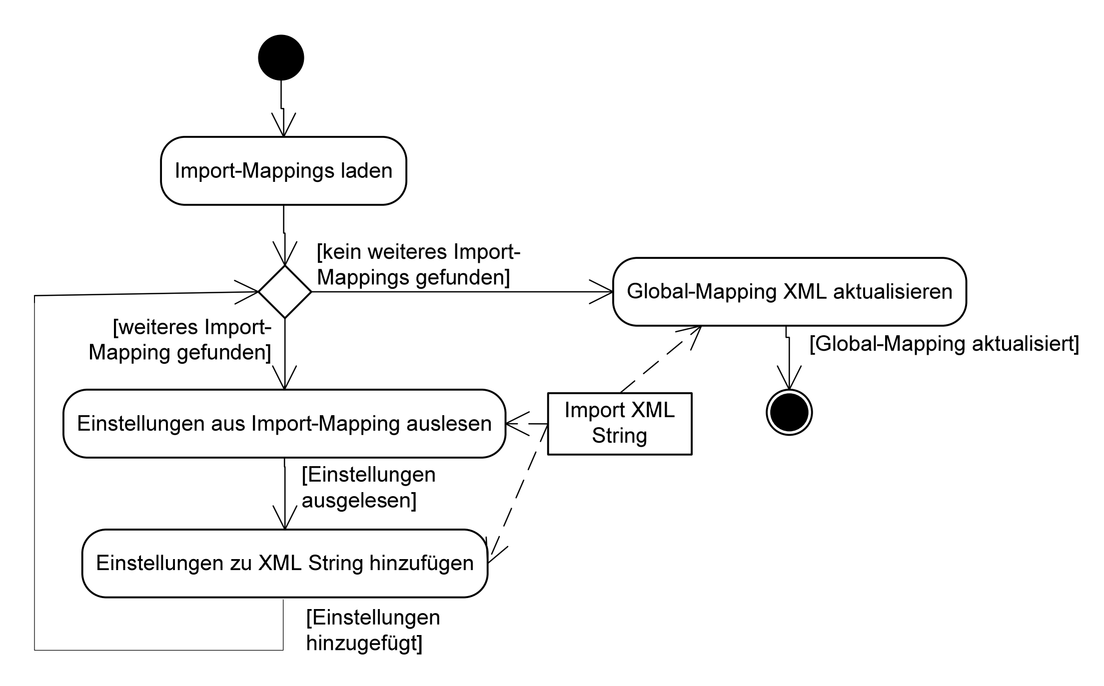


Abbildung 6-11: Ladeprozess für Global-Mapping

Das in Abbildung 6-11 dargestellte Aktivitätsdiagramm beschreibt den Ladeprozess des *Global-Mappings*. Beim Laden werden alle *Import-Mappings* des Projekts eingelesen und deren Einstellungen bzw. Daten in einen XML-Ausdruck geschrieben. Dieser Ausdruck bildet die Grundlage für die Vorschläge des *Global-Mappings*. Bei jedem neuen Aufruf des *Global-Mappings* wird dieser XML-Ausdruck neu erstellt.

6.5.3 Export-Komponenten

Die Export-Komponenten stellen die Schnittstellen zum SQL-MDi Query Parser und SQL-MDi Query Processor bereit. Um die Kommunikation mit dem SQL-MDi Query Parser zu ermöglichen muss an zwei unterschiedlichen Stellen angesetzt werden. Zum einen müssen die

Metadaten über die *Import-Schemen* und das *Global-Schema* exportiert und so aufbereitet werden, dass diese vom SQL-MDi Query Parser gelesen werden können. Zum anderen muss für jede Abfrage des SQL-MDi Query Parser eine SQL-MDi Datei zur Überbrückung von Heterogenität bereitgestellt werden. Das Werkzeug erstellt diese Datei durch den *Export SQL-MDi Wizard*.

Export Metadata Wizard

Um den Export der Metadaten zu bewerkstelligen, müssen vom *Wizard* alle im Projekt vorhandenen Schemata eingelesen und ausgewertet werden. Bei der Durchführung werden alle Modelle rekursiv durchlaufen. Währenddessen werden die Metadaten in Hash-Maps zwischengespeichert. Ein Zwischenspeichern der Elemente ist notwendig um nicht für jeden SQL Befehl separat eine Datenbankverbindung aufbauen zu müssen.

Beim Export sind einige Einschränkungen und Vorgaben zu beachten. So dürfen etwa mehrfach existierende Dimensionen nur einmal in der Datenbank gespeichert werden.

Die Speicherung der Metadaten erfolgt in zwei verschiedenen Datenbanken. Zum einem wird das Metadata-Dictionary erzeugt, welches die Information über alle verwendeten Schemata beinhaltet. Zum anderen wird das Dimension-Repository erzeugt, welches gesondert nochmals die Information zu den Dimensionen des Global-Schema aufbereitet.

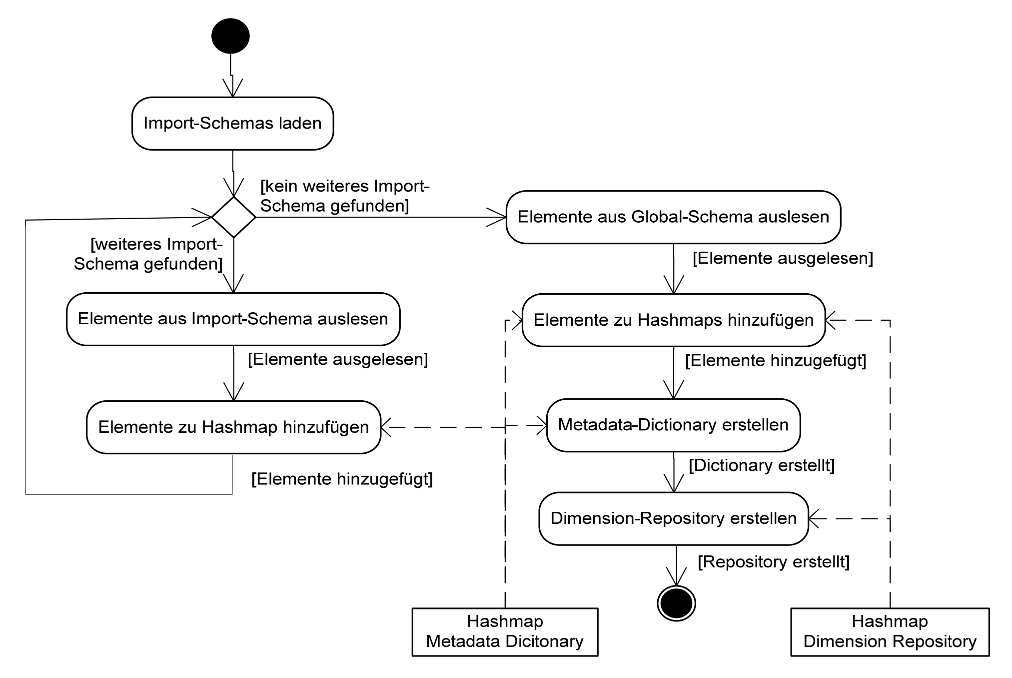


Abbildung 6-12: Aktivitätsdiagramm Export von Metadaten

In Abbildung 6-12 wird der Prozess des Exports von Metadaten anhand eines Aktivitätsdiagramms dargestellt. Im Diagramm werden besonders die unterschiedlichen Hash-Maps zur Speicherung der Elemente in den Metadata-Dictionary bzw. im Dimension-Repository hervorgehoben. Es wird explizit dargestellt, dass die Elemente des *Global-Schemas* im Metadata-Dictionary und im Dimension-Repository gespeichert werden müssen.

Export SQL-MDi Wizard

Das Resultat des *Export SQL-MDi Wizards* ist eine SQL-MDi Datei, die daraufhin beim SQL-MDi Query Parser als Eingabe verwendet werden kann.

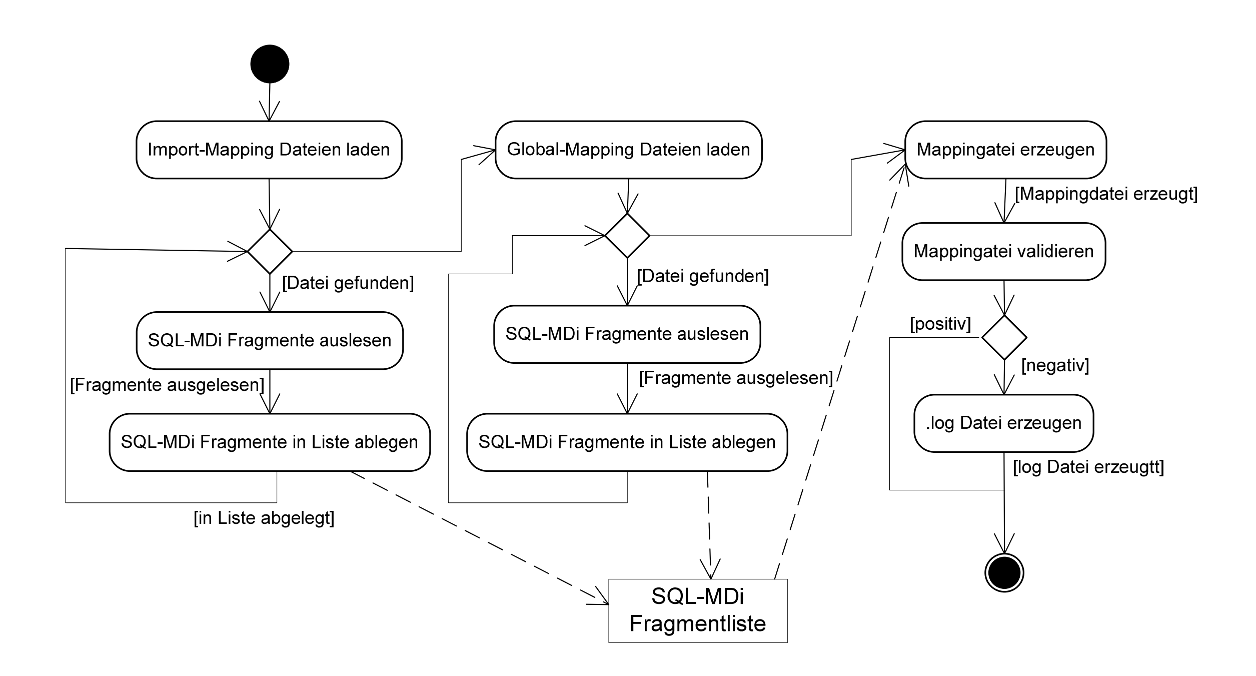


Abbildung 6-13: Funktionalität SQL-MDi Export

In Abbildung 6-13 wird der Export der SQL-MDi durch ein Aktivitätsdiagramms beschrieben. Zu Beginn werden die *Import-Mappings* eingelesen, da diese in der SQL-MDi Datei vorne angereiht werden müssen. Zur Zwischenspeicherung werden die SQL-MDi Fragmente in eine Liste gespeichert. Nach Traversierung der *Import-Mapping*-Dateien wird ein analoger Prozess mit den *Global-Mapping*-Dateien durchgeführt.

Sobald alle Mapping-Dateien des Projekts durchlaufen worden sind, wird die SQL-MDi Datei erzeugt. Hierbei werden alle Elemente einer Liste in eine Datei geschrieben. Nach dem Schreiben wird nochmals die syntaktische Korrektheit der Datei überprüft. Dazu wird der von

[Rossgatterer 2008, Brunneder 2008] entwickelte Validierungsmechanismus verwendet. Sollte dieser Mechanismus fehlschlagen wird eine *.log*-Datei mit den entsprechenden Fehlermeldungen erstellt.

6.6 Zusammenfassung

Aufbauend auf der in Teil I beschriebenen Theorie wurden in diesem Kapitel die Grundlagen für die Implementierung des Werkzeuges bereitgestellt. Für die Umsetzung wurde neben den von Eclipse geforderten Einschränkungen auch die im Metamodell vorgesehene Struktur angewandt.

Zu Beginn wurde die bei der Entwicklung angewendete Vorgehensweise des evolutionären Prototypings im Detail beschrieben. Anschließend wurden die verwendeten Technologien und die Systemarchitektur erläutert.

Die Systemarchitektur gliedert sich in die drei folgenden Komponentengruppen: schemabezogene Komponenten, Mapping-Komponenten und Export-Komponenten. Im Mittelpunkt der Architektur des Projektes steht das UML-Metamodell, welches als Basis für alle weiteren Tätigkeiten dient.

7 Systemimplementierung

In diesem Kapitel wird die Implementierung des Global Schema Architects erläutert, um Dritten die Möglichkeit zu geben, sich in das System einzuarbeiten und gegebenenfalls Änderungen und Erweiterungen durchführen zu können. Um einen besseren Vergleich zu ermöglichen gliedert sich die Beschreibung der Implementierung in gleiche Struktur wie die Systemarchitektur (siehe Abschnitt 6.5). Außerdem wird zu Beginn jeder Komponente der allgemeine Aufbau der Implementierung jeder Komponente beschrieben und in weiterer Folge werden in ausgewählten Komponenten Quelltext Ausschnitte explizit erläutert.

Zusätzlich wird in diesem Kapitel auf die konkrete Umsetzung der *plugin.xml*-Datei eingegangen, weil sie die Einbettung des Plug-ins in Eclipse regelt. In der Systemarchitektur wurde auf die Beschreibung der Datei *plugin.xml* verzichtet, da sie dafür nicht relevant ist.

7.1 Schemabezogene Komponenten

In diesem Abschnitt wird die Implementierung des Metamodells sowie aller in direktem Zusammenhang stehenden Komponenten beschrieben. Diese Kapitel baut auf der für diese Komponenten beschriebenen Architektur in Abschnitt 6.5.1 auf.

7.1.1 Komponente Metamodell

Die Beschreibung dieser Komponente teilt sich in drei Bereiche auf. Zum einen wird die Erstellung des Metamodells als Profil beschrieben. Zum anderen wird die Zusammenführung des Metamodellprofils mit einem Modell erläutert. Schlussendlich werden auf das Modell anwendbare Funktionen definiert.

Erstellung des Metamodells

Die Entwicklung des Metamodells erfolgte mit der von Eclipse bereitgestellten EMF basierenden Implementierung des UML2-Frameworks. Aufgrund des Einsatzes von EMF war für die Entwicklung dieses Modells keine explizite Codierungsarbeit notwendig.

Für die Umsetzung des Werkzeuges wurde das von Eclipse vorgegebene UML-Modell erweitert. Dies erfolgte durch Profile, welche als leichtgewichtige Erweiterungen zu UML-

Modellen angesehen werden. Profile können die Ausdruckstärke bereits vorhandener UML-Elemente durch Stereotypen erweitern. Dadurch können UML-Elementen zusätzliche Eigenschaften übergeben werden. Im Gegensatz dazu ist das Entfernen von Eigenschaften nicht möglich. Schwergewichtige Änderungen wie die Definition von neuen Modellen können mit diesem Ansatz nicht durchgeführt werden, was aufgrund einer Zielsetzung dieser Arbeit, nahe am UML-Standard zu bleiben, nicht notwendig ist. [Bruck & Hussey 2008]

Für die Erstellung eines Profils werden die von Eclipse bereitgestellten Assistenten verwendet. Der UML-Model Assistent erstellt eine UML-Datei. Profile werden üblicherweise mit der Endung „*.*profile.uml*“ benannt. Die Umsetzung des Gsa-Metamodells (siehe Abschnitt 5.3) findet sich in der Datei *Gsa.profile.uml* im Ordner *Model*. Die Umsetzung des Profils wird in Abbildung 7-1 dargestellt.

Durch das Profil werden alle Erweiterungen zum standardisierten UML bereitgestellt. Folgende Tabelle stellt die UML-Elemente mit den dazugehörigen Stereotypen dar.

Stereotyp	**UML-Element**
Cube (Würfel)	Package
Dimension	Package
Fact (Fakt)	Class
Level (Aggregationsebene)	Class
IdentifyingAttribute (identifizierendes Attribut)	Property
NonIdentifyingAttribute (nicht identifizierendes Attribut)	Property

Tabelle 7-1: Zuordnungen von Stereotypen

Vergleicht man Abbildung 5-5 und Tabelle 7-1 lässt sich die strikte Umsetzung des Metamodells erkennen. Der einzige Unterschied besteht in der Darstellung der Kenngrößen. In diesem Zusammenhang wurde auf die explizite Definition eines Stereotyps für *Measures* verzichtet. Dies begründet sich dadurch, dass in einem Fakt nur eine Art von Attribut (engl. *Property*) auftreten kann, somit muss nicht zwischen verschiedenen Attributen unterschieden werden können. Im Gegensatz dazu werden die Attribute einer Aggregationsebene in identifizierende und nicht identifizierende unterteilt.

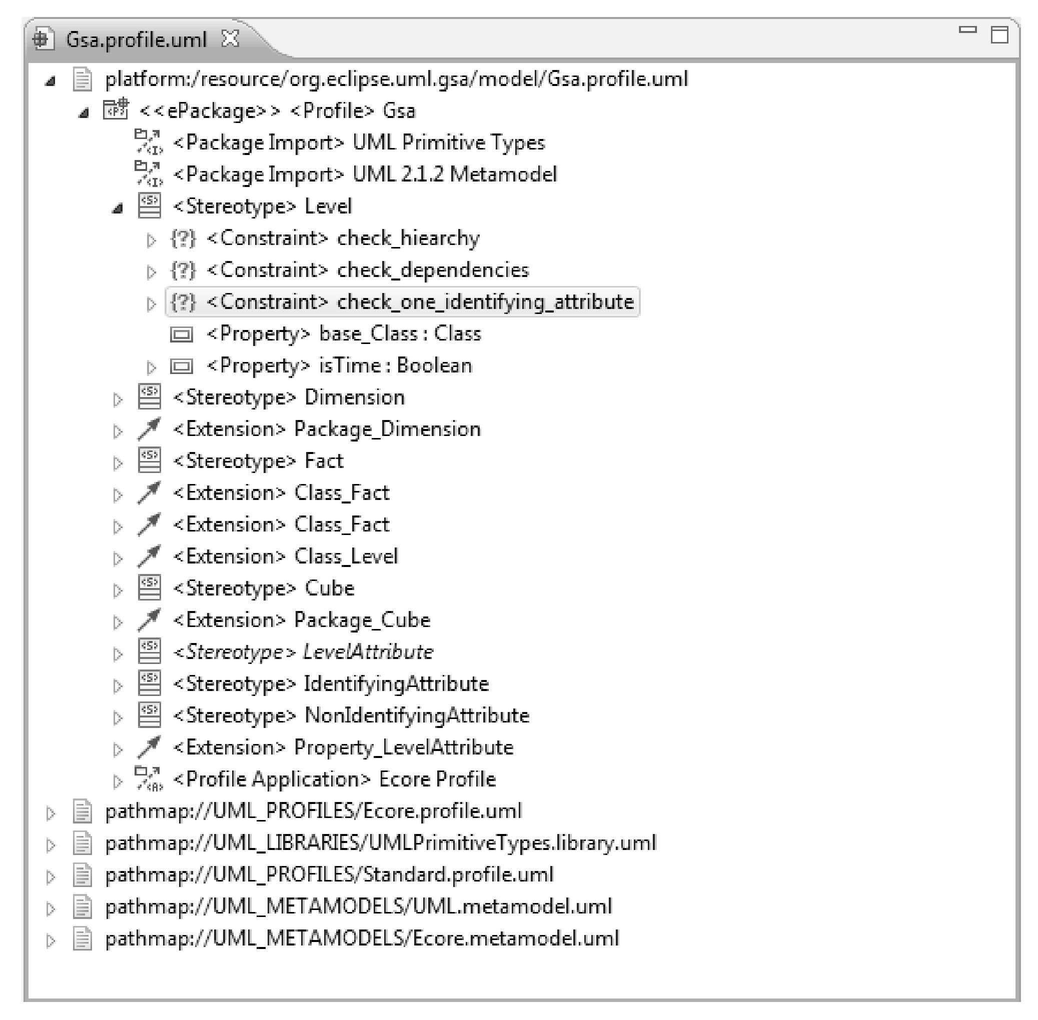


Abbildung 7-1: UML-Profil für Global Schema Architect

In Abbildung 7-1 wird das verwendete Profil dargestellt. Die Erstellung und Bearbeitung des Profils erfolgt ausschließlich über den in Abbildung 7-1 dargestellten UML-Editor. Für die Umsetzung des Profils in die endgültige Anwendung muss aus dem UML-Modell der notwendige Code generiert werden. Hierfür wird aus dem UML-Profil mithilfe eines sogenannten „genmodels“ der fertige Code erstellt. Der genaue Vorgang wird in einem Tutorial von [Bruck & Damus 2008] ausführlich erläutert.

Bei der Definition eines UML-Profils ist das Setzen von Constraints möglich. Mithilfe dieser Constraints können im Nachhinein Modelle hinsichtlich des korrekten Aufbaus überprüft werden. Constraints werden in der Object Constraint Language (OCL) spezifiziert. Mitthilfe dieser Notation können die anzuwendenden Regeln beschrieben werden. Die gesetzten Constraints werden folgend textuell als auch mit der OCL Notation beschrieben:

- **Würfel** (*Cube*) – *check_has_adequate_subelements*

  In jeden Würfel dürfen nur Dimensionen, Fakten oder Abhängigkeiten angeführt werden.

  - `self.nestingPackage->size = 0`
  - `self.nestedPackage->forAll (oclIsTypeOf(Dimension))`
  - `self.contents->forAll (oclIsTypeOf(Fact) or oclIsKindOf(Dependency)`

- **Dimension** – *check_has_levels*

  In Dimensionspaketen dürfen nur Aggregationsebenen (*Levels*) enthalten sein.

  - `self.contents->forAll (oclIsTypeOf(Level))`

- **Dimension –** *check_no_dependencies*

  Es werden keine Abhängigkeiten zwischen Dimensionen erlaubt.

  - `self.clientDependency->size = 0`

- **Fakt** *(Fact) – check_dependencies*

  Alle von einem Fakt ausgehenden Abhängigkeiten müssen auf eine Dimension gerichtet sein. Es muss mindestens auf eine Dimension verwiesen werden.

  - `self.clientDependency -> size >= 1`
  - `self.clientDependecy->forAll(supplier ->forAll(oclIsTypeOf(Dimension))`

- **Aggregationsebene** *(Level) – check_hierachy*

  Rekursive Abhängigkeiten zwischen den Aggregationsebenen einer Dimension sind nicht erlaubt.

  - `not self.allSuppliers->includes(self)`

- **Aggregationsebene** (Level) **–** *check_dependencies*

  Es dürfen nur Abhängigkeiten zwischen den Aggregationsebenen erstellt werden.

  - `self.clientDependency -> forAll (supplier -> forAll(oclIsTypeOf(Level))`

- **Aggregationsebene** (Level) – *check_one_identifying_attribute*
  Für jede Aggregationsebene darf nur ein Attribut als identifizierendes Attribut (*Identifying Attribute*) definiert werden.
  – `Self.ownedAttribute->select(oclIsTypeOf(IdentifyingAttribute))->size=1`

Der aus dem UML-Profil (siehe Abbildung 7-1) generierte Code wird automatisch in vier verschiedene Pakete unterteilt, wobei der Benutzer eigene Benennungen vergeben kann:

- `org.eclipse.uml.gsa.`**`gsa`**
- `org.eclipse.uml.gsa.`**`gsa.internal.impl`**
- `org.eclipse.uml.gsa.`**`gsa.internal.operations`**
- `org.eclipse.uml.gsa.`**`gsa.util`**

Der Entwickler kann diese Pakete als fertig und endgültig ansehen. Im Generellen ist das nachträgliche Editieren von generiertem Code nicht empfehlenswert, da dieser bei einer erneuten Generierung wieder überschrieben wird. Einzig im Paket **`org.eclipse.uml.gsa.gsa.internal.operations`** sind nachträgliche Anpassungen notwendig. Mit spezieller Kennzeichnung durch die Annotation *@generated = false* werden die Anpassungen beim nächsten Generieren nicht überschrieben. In diesem Paket werden u.a. Constraints definiert, welche von dem Benutzer selbständig implementiert werden müssen. Dieser Umstand ergibt sich daraus, dass EMF keine Möglichkeit bereitstellt, automatisch aus OCL Constraints Quelltext zu generieren. Zur Generierung von Quelltext aus OCL könnte der Ansatz Dresden OCL [Dresden OCL 2009] integriert werden. In dieser Arbeit wurde aufgrund des großen Umfangs nicht weiter auf diesen Ansatz eingegangen.

Um die Implementierung eines Constraints zu veranschaulichen wurde beispielhaft das Validieren der Hierarchie einer Dimension ausgewählt und in Listing 7-1 beschrieben. Die Methode `checkHierarchy()` ruft die Methode `checkCircles()` auf, welche von einer Aggregationsebene (engl. Level) ausgehend dieselbe wieder sucht. Wenn eine Aggregationsebene direkt oder über Umwege sich selbst referenziert, ist die Hierarchie einer Dimension demnach ungültig. Sollte dieser Fehler auftreten wird eine Fehlermeldung erstellt, der boolesche Wert „*false*“ zurückgegeben und mit der Validierung der anderen Elemente fortgefahren.

In Listing 7-1 werden zusätzlich erste Funktionalitäten zur Bearbeitung von UML-Dateien aufgezeigt. Eine explizite Beschreibung dieser Elemente erfolgt im weiteren Verlauf der Diplomarbeit.

```
public static boolean checkHierarchy(Level level,
        DiagnosticChain diagnostics, Map<Object, Object> context) {

    org.eclipse.uml2.uml.Class level_class = level.getBase_Class();
    boolean result = checkCircles(level_class, level_class);

    if (!result) {
        if (diagnostics != null) {
            diagnostics.add(new BasicDiagnostic(Diagnostic.ERROR,
                        GSAValidator.DIAGNOSTIC_SOURCE,
                        GSAValidator.LEVEL__HIEARCHY,
                        "No circles are allowed in the hierarchy of le-
vels",
                        new Object[] { level }));
        }
        return false;
    }
    return true;
}

private static boolean checkCircles(org.eclipse.uml2.uml.Class clazz,
        org.eclipse.uml2.uml.Class level_class) {
    for (org.eclipse.uml2.uml.Dependency dependency : clazz
            .getClientDependencies()) {
        for (org.eclipse.uml2.uml.NamedElement ne : dependency
                .getSuppliers()) {
            org.eclipse.uml2.uml.Class cl = (org.eclipse.uml2.uml.Class)
ne;
            if (cl.getName().equals(level_class.getName())) {
                return false;
            } else {
                return checkCircles(cl, level_class);
            }
        }
    }
    return true;
}
```

Listing 7-1: Validierung der Hierarchie in einer Dimension

Zusammenfügen von Profil und Modell

Das Anwenden eines Profils auf ein Modell ist ein zentraler Punkt dieser Arbeit. Durch diesen Arbeitsschritt werden Regeln und die Elemente des Profils auf das neue Modell übertragen. Die dafür notwendigen Arbeitsschritte werden in Listing 7-2 beschrieben.

```
//Create Model
Model gsaExample = createModel("global");

//Loading Profile
Profile myProfile = (Profile) loadProfile(URI.createURI("platform:/plugin/org.eclipse.uml.
gsa/model/").appendSegment(
"Gsa.profile").appendFileExtension(UMLResource.FILE_EXTENSION));

//Applying Profile
applyProfile(gsaExample, myProfile);
```

Listing 7-2: Zuweisung von Profil zum Model

Grundlage für die Verwendung der in Listing 7-2 beschriebenen Methoden ist das Erben von der `BaseGenerator` Klasse im `org.eclipse.uml.gsa.generators` Paket. Die genauen Funktionen dieser Klasse werden im nächsten Punkt ausführlich beschrieben.

Als ersten Schritt muss das `Model` mithilfe der Methode `createModel()` erzeugt werden. Um daraufhin das Profil (`Profile`) anzuwenden, muss auf die richtige Datei im Plug-in zugegriffen werden. Schlussendlich muss das Profil noch auf das Model angewandt (`applyProfile()`) werden.

Funktionen für Modelle

Das UML2-Framework von Eclipse stellt eine Vielzahl von Funktionen bereit mit welchen Modelle bearbeitet werden können. Diese Methoden müssen an mehreren Stellen im Projekt angewandt werden. Dem Prinzip „*Don't repeat yourself*" zufolge ist eine Auslagerung der Methoden auf eine höhere Ebene sinnvoll. Aus diesem Grund wurde die Klasse `BaseGenerator` eingeführt. Alle Klassen die direkt mit dem UML-Modell interagieren, müssen von dieser Klasse erben. Um eine übersichtliche Gestaltung zu gewährleisten sind alle Klassen die von `BaseGenerator` erben und mit dem Metamodell interagieren im Paket `org.eclipse.uml.gsa.generators` abgelegt.

Eine Ausnahme gilt für die Mapping-Editoren, welche zwar auf einem UML-Modell basierend, jedoch nicht das entsprechende Profil anwenden. Die von `BaseGenerator` abgeleiteten Klassen `NamedElementBase` und `MastercontentProvider` wurden zu den Paketen der jeweiligen Editoren hinzugefügt. Eine genaue Beschreibung wird in Abschnitt 7.2 gegeben.

Abbildung 7-2 zeigt ein Klassendiagramm mit `BaseGenerator` und dessen Unterklassen im Paket `org.eclipse.uml.gsa.generators`. Die genaue Funktionalität der jeweiligen Unterklassen wird in den entsprechenden Kapiteln näher erläutert.

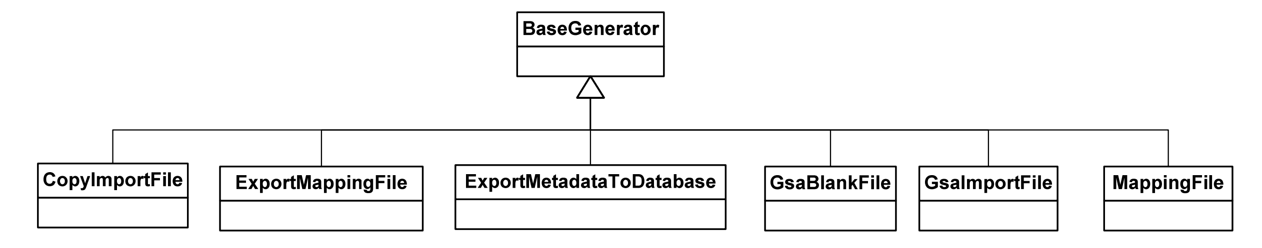


Abbildung 7-2: Klassendiagramm Generalisierung von BaseGenerator

Die `BaseGenerator` Klasse stellt u.a. Methoden zum Editieren von Modellen bereit:

- **`createPackage(Package, String, EClass)`:** Mit dieser Methode kann ein Paket erzeugt werden, auf das ein Stereotyp angewandt wird. Der Stereotyp wird in Form einer *EClass* als Parameter übergeben. Durch diese Methode können z.B. Würfel oder Dimensionen erzeugt werden. Der Parameter *Package* gibt jenes Paket an, in welches das anzulegende Paket inkludiert werden soll**.** Der *String* legt den Namen des zu erstellenden Paketes fest.
- **`createClass(Package, String, EClass)`:** Funktioniert analog zu `createPackage()`, mit dem Unterschied, dass eine Klasse erstellt werden kann. Anwendung findet diese Methode bei der Erstellung von Aggregationsstufen und Fakten.
- **`createAttribute(Package, String, EClass)`:** Erstellt ein Attribut inklusive eines darauf angewendeten Stereotypen.
- **`getDWElements(Package, LinkedList, String[])`:** Diese Methode ermöglicht bestimmte Elemente aus einem Paket auszulesen und einer Liste anzufügen. Es könnten beispielsweise alle Dimensionen und Levels ausgelesen werden. Diese Methode nimmt aufgrund der häufigen Anwendung bei der Implementierung des Global Schema Architects eine zentrale Rolle ein.
- **`applyAllStereotypes()`:** Fügt zu allen erzeugten Elemente die Stereotypen hinzu. Die Notwendigkeit einer solchen Methode ergibt sich daraus, dass ein Element gespeichert werden muss, bevor ein Stereotyp angewandt werden kann. Die oben beschriebenen „create“-Methoden speichern aus diesem Grund das Element und den dazugehörigen Stereotyp in einer speziellen Klasse (**`ElementStereotype)`** ab. Auf

Basis dieser Klasse können im Nachhinein die entsprechenden Stereotypen auf die Elemente angewandt werden.

7.1.2 Komponente Import-Schema

Die Komponente beinhaltet die für den Import eines Schemas und die Darstellung als UML-Modell notwendigen Klassen. Durch diese Komponente werden externe physische Datamarts die vom Global Schema Architect verlangte Struktur angepasst und importiert. Die Schwierigkeit in der Umsetzung des Import-Schemas liegt in den Heuristiken, welche aus den verschiedenen Tabellen eines Data Marts ein multidimensionales Modell bzw. einen Vorschlag für ein multidimensionales Modell bilden sollen. Dieser Prozess wurde in Abschnitt 6.5.1 bereits anhand eines Aktivitätsdiagramms dargestellt. In diesem Kapitel wird nun auf die konkrete Umsetzung eingegangen.

Für diese Komponente wird ein *Wizard* eingesetzt der den Benutzer bei der Eingabe der notwendigen Daten zur Datenbankverbindung unterstützt. Mittels einer zweiten *WizardPage* wird ein bearbeitbarer Vorschlag für die Erstellung des Import-Schemas bereitgestellt, demzufolge wird der Forderung der nachträglichen Editierbarkeit nachgekommen.

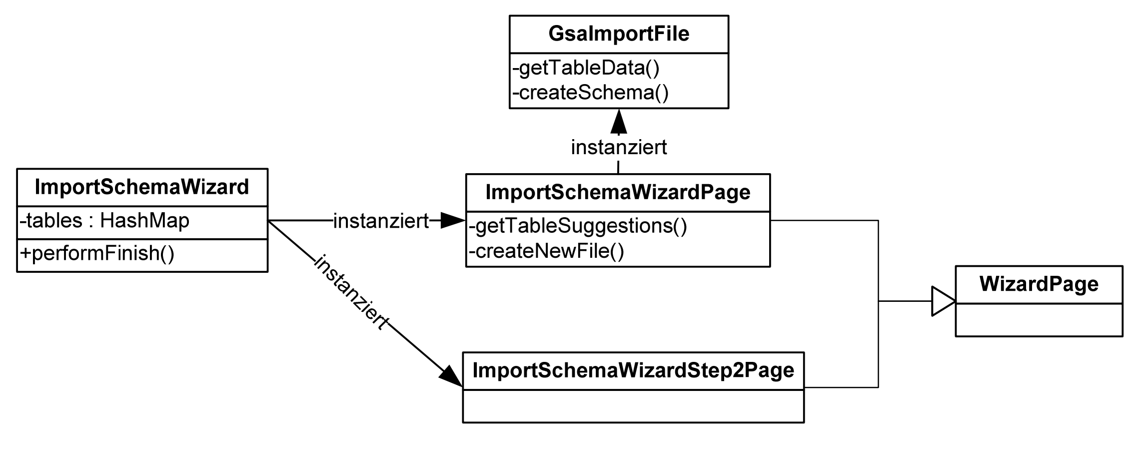


Abbildung 7-3: Aufbau Wizard Import-Schema

Abbildung 7-3 zeigt den grundsätzlichen Aufbau eines *Wizards* am Beispiel des Import-Schemas. Im Zentrum steht der `ImportSchemaWizard` welcher durch eine oder mehrere Unterseiten (*WizardPages*) ergänzt wird. Bei korrekter Eingabe der Werte in der `Import-SchemaWizardPage` wird mithilfe der Methode `getTableSuggestions()` automatisch ein Vorschlag für die endgültige Generierung des Schemas gegeben, wozu in weiterer Folge in der Klasse `GsaImportFile` die Methode `getTableData()` aufgerufen wird.

Dieser Vorschlag wird in Form einer Hash-Map im `ImportSchemaWizard` abgespeichert. Der Benutzer hat danach die Option das Ergebnis zu editieren, wofür die `ImportSchema-WizardStep2Page` verwendet werden kann. Sollte eine Änderung vorgenommen werden, wird diese vom `ImportSchemaWizard` direkt in der Hash-Map abgespeichert.

Bei Beenden des *Wizards* wird automatisch die `performFinish()` Methode aufgerufen, welche auf die `GsaImportFile` Klasse zugreift und daraufhin den Importvorgang einleitet. Als Basis für diesen Vorgang wird wiederum die Hash-Map aus dem `ImportSchemaWi-zard` verwendet.

```
public Hash-Map<String, String> getTableData() {
  int state = 0;
  if (!checkDbConnection()) return null;
  myTables = new Hash-Map<String, String>();
  if (conn != null) {
    DatabaseMetaData metadata;
    try {
      metadata = conn.getMetaData();
         ResultSet tables = metadata.getTables(null, dbds.getSchema(),
"%",
          new String[] { "TABLE" });
      while (tables.next()) {
        String tablename = tables.getString("TABLE_NAME");
        ResultSet foreignKeys = metadata.getExportedKeys(null,
            dbds.getSchema(), tablename);
        //Dimension
        while (foreignKeys.next()) {
          String fkName = foreignKeys.getString("FKCOLUMN_NAME");
          myTables.put(tablename, "DIMENSION");
        }
        ResultSet foreignKeys2 = metadata.getImportedKeys(null,
            null, tablename);
        //Fact
        while (foreignKeys2.next()) {
          myTables.put(tablename, "FACT");
          break;
        }
        //Tables which don't fit to any scheme
        if (!myTables.containsKey(tablename)) {
          myTables.put(tablename, "NONE");
        }
      }
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
  return myTables;
```

Listing 7-3: Erstellung des Vorschlages für den Import

Listing 7-3 beschreibt den Quelltext für den Schema-Vorschlag. Die Notwendigkeit zur Beschreibung dieses Listings ergab sich aus der Komplexität der Methode. Durch eine genaue Beschreibung soll die Verständlichkeit erhöht werden. Dieser Algorithmus wurde bereits in der Systemarchitektur in Abschnitt 6.5.1 mittels eines Aktivitätsdiagramms dargestellt. Im Vergleich zum letztendlichen Quelltext wurde in diesem Beispiel aufgrund der besseren Lesbarkeit auf unbedeutende Codestellen verzichtet. In den ersten Zeilen des Listings wird die Datenbankverbindung überprüft und die Metadaten werden eingelesen. In Zeile 11 werden die ausgelesenen Tabellen traversiert. Daraufhin werden die exportierten Fremdschlüssel ermittelt. Sollte eine Tabelle solche enthalten, kann diese als Dimension definiert werden. Sollten importierte Fremdschlüssel vorhanden sein, wird die Tabelle als Fakt definiert. Im Falle nicht vorhandener Schlüssel wird die Tabelle mit dem Wert „*NONE*“ maskiert.

7.1.3 Komponente Global-Schema

Das Global-Schema wird entweder als leeres Modell erzeugt oder aus einem existierenden Import-Schema dupliziert. Zusätzlich werden vom Benutzer bei der Erstellung des Global-Schemas die notwendigen Daten zur Verbindung zum Dimension-Repository verlangt. Die Weiterverarbeitung der Kopie oder des leeren Modells erfolgt mit dem grafischen Editor oder direkt am Modell.

Um ein leeres Global-Schema zu erstellen wird durch den `GsaBlankWizard` die Klasse `GsaBlankFile` aufgerufen die ein leeres Modell, auf das bereits das Profil des GSA-Metamodells angewandt wurde (siehe Abschnitt 7.1.1), erzeugt. Die vom Benutzer eingegebenen Daten zur Speicherung des Global-Schemas können optional durch die Methode `checkDbConnection()` vom Benutzer überprüft werden.

Der `GsaFromImportSchemaWizard` erzeugt die Klasse `CopyImportFile`. Diese Klasse erstellt eine Kopie eines importieren Schemas. Beim Kopieren der Datei werden die Verbindungsdaten des Import-Schemas durch die vom Benutzer eingegebenen Daten überschrieben. Eine Kopie eines Schemas erfolgt über das Duplizieren aller beinhaltenden Elemente. Um dies zu ermöglichen wurde die Methode `duplicatePackage()` eingeführt. Diese kopiert ausgehend vom Wurzelpaket des Import-Schemas alle Elemente. Um sämtliche Ebenen des Metamodells abzudecken, wird diese Methode rekursiv aufgerufen. Sollte z.B.

das Wurzelpaket ein weiteres Unterpaket enthalten, wird die Methode nochmals mit dem Unterpaket als Parameter aufgerufen.

7.1.4 Grafischer Editor für Schemakomponenten

Der grafische Editor basiert auf dem GMF (Graphical Modelling Framework). Das GMF unterstützt die automatische Generierung von GEF-Code aus EMF-Modellen (siehe Abschnitt 4.4). Im Endeffekt entsteht ein GEF-Editor [GMF 2009]. Der Einsatz des grafischen Editors ergibt sich aus der Forderung einer benutzerfreundlichen Bedienung, die mit dem standardmäßigen UML-Modell Editor nicht gegeben ist. Durch den grafischen Editor können die einzelnen Elemente des Metamodells durch einfaches Ziehen und Klicken erstellt und bearbeitet werden.

Für eine fundierte Erklärung des GEF-Editors ist es wichtig auf die ihm zugrunde liegenden Patterns und Designmethoden einzugehen. Der GEF-Editor basiert auf dem MVC (Model View Controller) Framework und verwendet das Composite Pattern.

In weiterer Folge werden der Aufbau und die genaue Anbindung des Editors im Detail beschrieben sowie auf für dieses Projekt notwendigen Änderungen eingegangen.

Model View Controller

Das MVC-Framework wurde erstmals von der Programmiersprache Smalltalk verwendet und behandelt die drei wesentlichen Komponenten einer GUI (Graphical User Interface). Das MVC ermöglicht durch die Verteilung der Aufgaben eine höhere Wiederverwendbarkeit und eine einfachere Anpassung. Das *Model* repräsentiert die Datenstrukturen und ist von *View* und *Controller* unabhängig. Die Anzeige erfolgt mittels der dazugehörigen *View,* welche in der Regel auf *Controller* und *Model* zugreifen kann. Durch den Controller werden Benutzerinteraktionen aufgenommen und die dazugehörigen Aktionen auf *View* und *Model* weitergeleitet [Pomberger & Blaschek 1996 S. 257f].

In vielen Programmiersprachen und Konzepten wird das MVC an die spezifischen Bedürfnisse angepasst. Beim GEF wurde die Beziehung zwischen *Model* und *View* entfernt. Somit übernimmt der *Controller* den kompletten Datenfluss. Außerdem wurden im GEF neue Namenskonventionen vorgegeben. Der *Controller* wird in diesem Zusammenhang als *EditPart*

und die *View* als *Figure* bezeichnet. In Abbildung 7-4 wird der genaue Aufbau des MVC-Konzepts im Zusammenhang mit GEF grafisch dargestellt [Gruhn et al. 2006 S. 289].

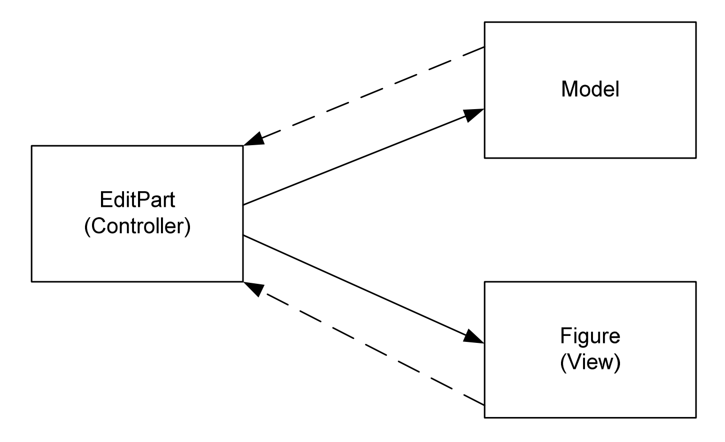


Abbildung 7-4: MVC für GEF angelehnt an [Gruhn et al. 2006 S. 290]

Composite Pattern

Durch die Verwendung des Composite Pattern soll dem Benutzer die Möglichkeit gegeben werden aus mehreren kleinen Elementen ein größeres Element zu formen, welches wieder mit mehreren größeren Elementen verbunden werden kann. Dies bedeutet, dass rekursive Beziehungen zwischen einzelnen Elementen abgebildet werden können. Das Grundelement eines Composite Patterns ist eine Klasse, die den Container sowie die primitiven Datentypen abbilden kann.

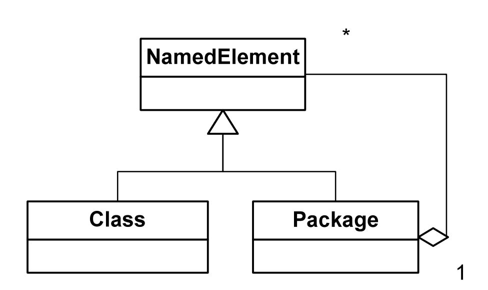


Abbildung 7-5: Composite Pattern für grafischen Editor

Abbildung 7-5 zeigt eine beispielhafte Anwendung des Patterns im Global Schema Architect. Durch Anwendung des Patterns ist es möglich die im Metamodell vorgesehene Paketstruktur zu erreichen. Beispielweise wird ein Würfel als Paket (engl. *Package*) dargestellt und kann wiederum mehrere Pakete (Dimensionen) und Klassen (Fakten) enthalten. Die Pakete der Dimensionen wiederum enthalten mehrere Klassen (Aggregationsebenen), dadurch kann die Struktur hierarchisch in mehrere kleine Unterelemente zerlegt werden.

Aufbau und Anbindung an den Global Schema Architect

Der Editor ist Teil der Eclipse UML2Tools (siehe Abschnitt 4.4.2) und wird als erweiterndes Plug-in in den Global Schema Architect integriert. Die UML2Tools stellen Editoren für alle Bereiche der UML bereit. Für den Global Schema Architect ist aufgrund der in Kapitel 5.1 gestellten Anforderungen ausschließlich das Klassendiagramm von Bedeutung.

Der Editor setzt auf dem UML-Modell auf und stellte neue Editierungsmöglichkeiten bereit. Bei der Durchführung einer Aktion im Editor werden die Änderungen direkt im UML-Modell gespeichert. Bezogen auf das MVC-Modell wird das UML-Modell demnach als *Model* des grafischen Editors verwendet. Jegliche Änderungen die durch den Editor durchgeführt werden, können auch am Modell direkt, allerdings in grafisch weniger ansprechender Baumdarstellung, durchgeführt werden. Die Unterscheidung der verschiedenen Dateien lässt sich durch die Namenskonventionen für die Dateitypen, **.umlclass* für den grafischen Editor und **.uml* für das Modell, erkennen.

Für eine adäquate Integration war es notwendig den Editor zu erweitern. So mussten z.B. die für das Metamodell notwendigen Elemente, wie Würfel und Dimension, in das Auswahlfeld des Editors hinzugefügt werden. Dazu wurden in der Klasse `UMLPaletteFactory` im Paket `org.eclipse.uml2.diagram.clazz.part` die entsprechenden Schaltflächen integriert. Eine Schwierigkeit bei der Erstellung der Elemente war das Hinzufügen der entsprechenden Stereotypen. Diese Problematik ergibt sich aus dem Grund, dass einem Element erst nach seiner Speicherung ein Stereotyp zugewiesen werden kann. Zu diesem Zweck wurde eine statische Variable (`UMLPaletteFactory.stereotype`) eingeführt, welche den Stereotypen für den Zeitraum zwischen Auswahl des Elements aus der Werkzeugliste und dem endgültigen Anfügen des Elements zwischenspeichert. Ebenso werden zu diesem Zeitpunkt auch die passenden Repräsentationsbilder (*Icons*) für das jeweilige Element eingefügt. Die Anpassung der Bilder und Stereotypen erfolgt im `EditPart` zum jeweiligen Element, das sich im Paket `org.eclipse.uml2.diagram.clazz.edit.parts` befindet. Durch den `EditPart` wird die `doDefaultElementCreation()` Methode des entsprechenden `Commands` aufgerufen. Zur Erzeugung einer als Paket repräsentierten Dimension, wird z.B. die Klasse `PackageCreateCommand` durch die Klasse `PackageEdiptPart` aufgerufen.

7.2 Mapping-Komponenten

Die Mapping-Komponenten erstellen die für die Ausführung des SQL-MDi Query Processors als Eingabe dienende SQL-MDi Datei. Der Aufbau der Mapping-Komponenten orientiert sich an der in Abschnitt 6.5.1 beschrieben Architektur. Um dem Benutzer eine möglichst einfache Bedienung zu ermöglichen, wurde der Mapping-Editor erstellt. Die Mapping-Editoren für das *Import*- als auch das *Global-Mapping* verwenden dasselbe Grundgerüst. Darum wird zu Beginn auf den grundlegenden Aufbau der Mapping-Editoren und in weiterer Folge auf die speziellen Eigenheiten der einzelnen Komponenten eingegangen.

Der Mapping-Editor teilt sich in mehrere Unterseiten, womit dem Benutzer ein höchstmöglicher Komfort geboten wird. Durch unterschiedliche Editorseiten sind mehrere Sichtweisen auf die gleiche Problematik möglich. Die Editoren gliedern sich in drei Unterseiten:

- **Schema Editor**: Dies ist die Standardseite des Editors und für den Großteil der Anwendungsfälle die adäquate Ansicht. Sie ermöglicht es dem Benutzer die gewünschten Änderungen vorzunehmen.
- **Code**: Diese Seite gibt dem Benutzer die Möglichkeit den Quelltext des Mappings direkt zu bearbeiten. Dies ist jedoch nur für Experten zu empfehlen.
- **Preview**: Diese Ansicht gibt dem Benutzer einen Überblick über den erstellten SQL-MDi Code.

Für die Umsetzung der einzelnen Editorseiten wurden Eclipse Forms eingesetzt, die im Folgenden näher erläutert werden.

Eclipse Forms

Eclipse Forms versuchen durch den Einsatz des SWT Frameworks den typischen „*Web look*“ in einer Eclipse-Applikation zu simulieren. Die verschiedenen Hauptelemente von Eclipse (Editoren (engl. Editors), Sichten (engl. Views), Assistenten (engl. Wizards) und Dialoge) sollen zur Verbesserung des „*look and feels*“ mithilfe der Eclipse Forms einheitlich repräsentiert werden.

Die Eclipse Forms unterstützen eine Vielzahl aus dem Web bekannter Formularelemente wie Textfelder, Passwortfelder, Schaltflächen, Auswahlfelder etc. Außerdem werden verschiedene Layout-Methoden wie Gridlayout (gitterförmige Anordnung), Tablelayout (tabellenförmige Anordnung) oder Borderlayout (komponentenbasierte Ausrichtung) bereitgestellt um eine optimale Ausrichtung der Symbole zu erreichen. [Glozic 2005]

Master Detail Block

Der Master Detail Block wird in der grafischen Benutzeroberfläche der Mapping-Editoren (d.h. *Schema Editor*) eingesetzt und basiert auf dem Master-Detail Pattern. Die Funktionsweise dieses Patterns ergibt sich aus einer Liste (Master) und einem Einstellungsbereich für jedes der Listenelemente (Detail). Die grafische Repräsentation erfolgt mit den Eclipse Forms welche die Anzeige in zwei Bereiche teilen. Auf der linken Seite findet sich eine Liste mit allen vorhandenen Elementen und auf der rechten Seite finden sich die Details zu diesen Elementen. [Betsy et al. 2009]

Auf ein föderiertes DW bezogen werden in der Mastertabelle alle Elemente eines Schemas (Fakten, Dimensionen etc.) aufgelistet. Im Detailbereich können zu den jeweiligen Elementen die Mappings definiert werden.

Umsetzung Mapping-Editor

Im Zentrum der Implementierung steht die Klasse `MulitPageEditor` bzw. `GsaMultiPageEditor` im Paket `org.eclipse.uml.gsa.editors`. In dieser Klasse werden die einzelnen Unterseiten des Editors erstellt und mit der `addPages()` Funktion dem Editor hinzugefügt. Zusätzlich behandelt diese Klasse die Speicherung sowie die komplette Steuerung des Editors.

Der Master Detail Block wird für die Umsetzung der *Schema Editor*-Seite des Mapping Editors angewandt. Im Mittelpunt des *Schema Editors* steht die Klasse `ScrolledPropertiesBlock`. Diese liest die notwendigen Daten ein, erzeugt und füllt den Masterbereich. Hierbei werden den Elementen der Mastertabelle die jeweiligen Detailseiten zugewiesen.

Die Mastertabelle beinhaltet pro Element des bearbeiteten Schemas eine *Typ*Klasse *(engl. Type)* für die Zugriffsfunktionen auf das Modell sowie eine *Details*-Klasse für die grafische Repräsentation der Einstellungen und Funktionen für das Element. Ein Typ entspricht jeweils einer Metaklasse im Metamodell (siehe Abbildung 5-5). Durch die Methode `save()` werden Einstellungen zu einem Typ im Modell abspeichert. Demnach existiert für jeden Typ eine eigene `save()` Methode. Die Speicherung der jeweiligen Einstellungen zu den Instanzen eines Typs wird in Form eines XML-Ausdrucks bei dem entsprechenden Element des Mapping-Schemas gespeichert.

Jeder für die Repräsentation der Metaklassen verwendete Typ erbt von der Klasse `NamedElementBase`, welche wiederum von der `BaseGenerator` Klasse erbt. Durch diesen Aufbau kann jeder Typ mit einem UML-Modell kommunizieren. Die `NamedElementBase` Klasse stellt notwendige Methoden und Funktionalitäten für die Unterklassen bereit. Beim Erstellen eines Objekts der Klasse `NamedElementBase` wird überprüft, ob die angegebene Datenquelle (ein Element des UML-Modells) bereits einen XML-Ausdruck enthält. Sollte das der Fall sein wird dieser automatisch geladen. Ansonsten wird ein leerer String erzeugt. Weiters befinden sich in der Klasse `NamedElementBase` Methoden zum Setzen und Auslesen des XML-Ausdrucks, sowie zur Speicherung der Änderungen im UML-Modell.

Grundlage jeder Detailseite ist die Klasse `BasicDetailPage`. Diese Klasse legt die Darstellung der verschiedenen Elemente fest und stellt weitere Methoden zum Ein- und Auslesen von XML-Elementen bereit. Zusätzlich enthält diese Klasse drei abstrakte Methoden, die durch jede Unterseite implementiert werden müssen:

- `createSettingsString()`: Erzeugt eine XML-Repräsentation der Einstellungen der Detailseite. Für jedes Detailelement muss eine Struktur für die Speicherung der Daten definiert werden.
- `generateSqlMdiText()`: Aufgrund der eingegeben Daten muss jede Detailseite den SQL-MDi Code erzeugen können, der später in der Vorschau und im endgültigen Dokument angezeigt wird. In Ausnahmefällen kann auf eine Implementierung dieser Methode verzichtet werden, z.B. wenn das übergeordnete Element auch den Ausdruck des untergeordneten Elements erzeugt.

- `update()`: Durch den Aufruf einer einzelnen Methode müssen die einzelnen Elemente einer Detailansicht aktualisierbar sein. In dieser Methode muss demnach die ganze Detailseite neu geladen werden können.

Durch die Einführung der beschriebenen Methoden ist eine automatisierte Abarbeitung aller Detailseiten möglich. Ein konkretes Beispiel hierzu wird in Abschnitt 7.2.1 gegeben.

7.2.1 Komponente Import-Mapping

Zur Erstellung des Import-Mappings werden das Global-Schema und das Import-Schema herangezogen. Das Import-Schema wird als Basis für das Import-Mapping vorgegeben. Im Gegensatz dazu wird das Global-Schema zum Erstellen von Vorschlägen bei der Erstellung der Mappings herangezogen.

Die Erstellung eines Vorschlags wird folgend beispielhaft dargestellt. Aufbauend auf das Rahmenbeispiel will der Benutzer den Würfel MFA (siehe Abbildung 1-2) dem globalen Würfel zuordnen. Dazu muss der Benutzer u.a. die Dimensionen angeben, welche auf das globale Schema zugeordnet werden sollen. Durch den Editor erhält der Benutzer einen Vorschlag über die vorhandenen Dimensionen. So kann etwa Tippfehlern und anderen Eingabefehlern vorgebeugt werden. Nach Auswahl des Benutzers werden die ausgewählten Dimensionen vom System gespeichert.

Beim Erstellen eines Import-Mappings liest die `getElements()` Methode der `MasterContentProvider` Klasse, welche eine private Klasse der `ScrolledPropertiesBlock` Klasse ist, alle Elemente aus dem Import-Schema ein und erzeugt die passenden Einträge in der Detail-Tabelle. Die jeweilige Detailklasse wird mit dem ersten Aufruf von einem der Elemente der Mastertabelle erzeugt.

Die Erstellung eines Vorschlags für auf das Global-Schema zuordenbare Dimensionen erfolgt durch die Methode `handleBrowseReferenceDimensions()`, die in der Klasse `TypeCube` die Methode `getOwnedElements()` aufruft. Daraufhin wird wiederum die zentrale Methode `getDWElements()` mit dem Parameter „Gsa::Dimension“ aufgerufen. Abbildung 7-6 stellt den Zusammenhang der Methoden anhand eines UML-Klassendiagramms dar.

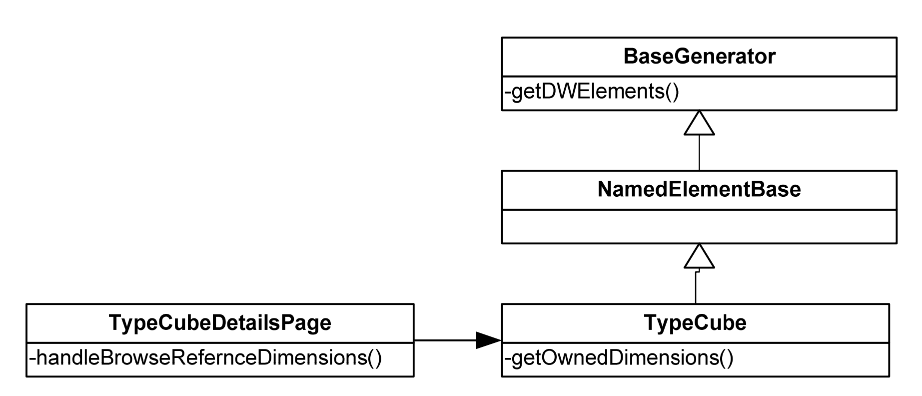


Abbildung 7-6: Auslesen der vorhandenen Dimensionen

Die ausgewählten Dimensionen werden in der Tabelle `dimensionsTable` zwischengespeichert. Aufgrund der zentralen Position in der Klasse ist die Tabelle für das Verständnis des Listing 7-4 notwendig. In Zeile 7 werden die Elemente aus der Tabelle ausgelesen.

Beim Speichervorgang der Detailseite werden automatisch die drei vorgegebenen Methoden (`createSettingsString ()`, `generateSqlMdiText()` und `update()`) der Basisklasse ausgeführt. Diesen Methoden haben aufgrund der zentralen Rolle eine wichtige Bedeutung für das Verständnis des Mapping-Editors. Sie zeigen auf wie die XML-Ausdrücke sowie die SQL-MDi Ausdrücke erstellt werden. Nachstehend wird, zur besseren Illustration, die Implementierung der Methoden in der Klasse `TypeCubeDetailsPage` als Beispiel herangezogen.

```
01 protected String createSettingsString() {
02      StringBuffer sb = new StringBuffer();
03      sb.append("<alias>");
04      sb.append(cube_short.getText());
05      sb.append("</alias>");
06      sb.append("<dims>");
07      TableItem[] dimItems = dimensionsTable.getItems();
08      for (TableItem item : dimItems) {
09              sb.append("<dim>");
10              sb.append(item.getText());
11              sb.append("</dim>");
12      }
13      sb.append("</dims>");
14      sb.append("<measures>");
15      TableItem[] factItems = factsTable.getItems();
16      for (TableItem item : factItems) {
17              sb.append("<measure>");
18              sb.append(item.getText());
19              sb.append("</measure>");
20      }
21      sb.append("</measures>");
22
23      return sb.toString();
24 }
```

Listing 7-4: Erzeugen von XML Settings-String

Listing 7-4 beschreibt eine beispielhafte Erzeugung des XML-Ausdrucks. Im Fall von `TypeCubeDetailsPage` werden die verwendeten Dimensionen und Kenngrößen in den XML-Ausdruck überführt. Die zuvor in der Tabelle `dimensionsTable` gespeicherten Werte werden nun ausgelesen und von Tags umgeben als XML-String zurückgegeben.

```
protected String generateSqlMdiText()
{
	StringBuffer sb = new StringBuffer();
	...
	LinkedList<String> items = new LinkedList<String>();
	LinkedList<String> dimItems = input.getDimensions();
	LinkedList<String> factItems = input.getMeasures();
	items.addAll(factItems);
	items.addAll(dimItems);
	...
	if (items.size() > 0) {
		sb.append("(");
		for (int i = 0; i < items.size(); i++) {
			if (i > factItems.size() -1 && i < (factItems.size() + dimI-
tems.size())) {
				sb.append("Dim ");
				sb.append("<dim#"+items.get(i)+"#>");
			}
			...
		}
		sb.append(")");
	}
	sb.append(input.getLowerLevelSqlMdi());
	return sb.toString();
}
```

Listing 7-5: Erzeugen des SQL MDi Ausdruckes

Listing 7-5 erläutert das Erzeugen des SQL-MDi Ausdruckes. Aufgrund der Komplexität wird auf die Erklärung von nicht notwendigen Codefragmenten verzichtet. Die `input` Variable in dieser Methode ist eine Klassenvariable, welche die `TypeCube` Datei referenziert. In dieser Methode werden zu Beginn die relevanten Daten aus den XML-Ausdrücken ausgelesen. Daraufhin werden aus den Daten die SQL-MDi Ausdrücke erstellt. Der Ausdruck `<dim# .. #>` steht in diesem Zusammenhang für einen Platzhalter, der bei der Vorschauansicht bzw. beim Export wieder ersetzt wird. Am Ende wird die Methode `getLowerLevelSqlMdi()` aufgerufen. Diese greift auf die in der Hierarchie untergeordneten Elemente zu und liest dort den entsprechenden SQL-MDi Code aus.

7.2.2 Komponente Global-Mapping

Das Global-Mapping referenziert im Gegensatz zum Import-Mapping nur das Global-Schema, liest jedoch beim Erstellen der Datei auch die im Projekt enthaltenen Import-

Mappings ein. Dadurch kann die Anforderung, alle bereits im Import-Mapping vorgenommenen Anpassungen im Global-Mapping zu berücksichtigen, erfüllt werden. Das Global-Mapping passt sich demnach den Änderungen im Import-Mapping an. Die jeweiligen Anpassungen werden in Form von XML-Ausdrücken aus den verschieden Import-Mappings eingelesen und zu einem XML-Ausdruck verbunden (siehe Abbildung 6-11). Der Ausdruck wird bei jedem Aufruf des Global-Mappings neu geladen.

Ein weiterer großer Unterschied zwischen Import-Mapping und Global-Mapping ergibt sich aus der fixen Anzahl der in der Mastertabelle repräsentierten Elemente beim Import-Mapping. Im Gegensatz dazu werden diese beim Global-Mapping dynamisch erzeugt (*Merge Dimensions* und *Merge Cubes*). Dies bedeutet, dass die notwendigen Daten aus den Import-Mappings bei jedem Aufruf des Global-Mappings aktualisiert werden.

Das Einlesen der Einstellungen für das Global-Mapping erfolgt durch die Methode `getElements()` in der privaten Klasse `MasterContentProvider` welche sich in der Klasse `ScrolledProperitesBlock` befindet. Aufgrund der Komplexität des XML-Ausdrucks wird xPath zum Auslesen der verschiedenen Elemente eingesetzt. Diese Technologie vereinfacht das Auslesen von Information.

```
//cube/id[text()='c1']/..//dimension/name
```

Listing 7-6: Beispiel XPath Abfrage

Die in Listing 7-6 beschriebene Abfrage aller Dimensionsnamen in einem Würfel zeigt mit welch geringem Aufwand eine xPath-Anfrage die notwendige Information bezieht. Bei Verwendung eines SAX- oder DOM-Parsers wäre eine Vielzahl von Iterationen notwendig um dasselbe Ergebnis zu erzielen.

Um der Mastertabelle dynamisch Elemente hinzufügen zu können, musste neben der `getElements()` Methode, welche die bereits vorhandenen Elemente beim Laden einliest, eine weitere Funktionalität eingeführt werden. Dahingehend wurden Funktionen zum Hinzufügen von *Merge Dimensions* und *Merge Cubes* Anweisungen erstellt. Die Speicherung der dynamischen Elemente erfolgt als Klasse innerhalb des UML-Modells für das Mapping. Durch die Speicherung als Klasse können die Elemente beim nächsten Aufruf des Editors einfach wie-

der geladen werden. Listing 7-7 beschreibt den Vorgang des Erzeugens eines neuen *Merge Dimensions* Tabellenelements.

```
String text = "Merge Dimensions "+countMergeDimensions;
TypeMergeDimensions tmd = new TypeMergeDimensions(uri, null, root, rootnew, text);
TableItem item = new TableItem(t, SWT.NULL);
item.setData(tmd);
item.setText(text);
```

Listing 7-7: Erzeugen eines neuen Tabellenelements

Die Besonderheit an Listing 7-7 ist das Erzeugen einer Instanz der `TypeMergeDimensions` Methode, wobei der angeführte `null` Wert näher erläutert werden sollte. Dieser Parameter wird üblicherweise für eine schon vorhandene Klasse (z.B. beim Starten des Editors) verwendet. Da in diesem Fall jedoch eine neue Klasse erzeugt werden soll, kann diese folglich noch nicht übergeben werden. Im Konstruktor der Klasse `TypeMergeDimensions` wird daher eine neue Klasse erstellt und durch Speichern in das UML-Modell übernommen.

7.3 Export-Komponenten

Durch Export-Komponenten sollte eine Interaktion mit den anderen Programmen (SQL-MDi Query Parser und SQL-MDi Query Processor) ermöglicht werden. Die Implementierung der einzelnen Export-Komponenten bezieht sich auf die in Abschnitt 6.5.3 beschriebene Architektur. Dies funktioniert über das Bereitstellen von Daten, welche für die Ausführung der anderen Programme notwendig sind.

7.3.1 Komponente Export Meta-Data

Um Meta-Daten zu exportieren, werden alle im Projekt vorhandenen UML-Modelle (Namenskonvention für Dateiname: **.import.uml* oder **.gsa.uml*) eingelesen und darauf aufbauend die notwendigen SQL-Ausdrücke erstellt. Während dieses Prozesses werden nur die Daten und nicht die Tabellenstruktur exportiert, weshalb das Metadata-Dictionary schon vor dem Export die passende, vom SQL-MDi Query Parser geforderte Struktur aufweisen muss [Brunneder 2008]. Die Verantwortung für die Erstellung der entsprechenden Struktur liegt beim Systemadministrator. Die Auswahl der Zieldatenbank (Metadata-Dictionary) kann der Benutzer mittels des Assistenten definieren. Üblicherweise wird diese Datenbank als „*OLAP_Repository*" benannt. Zusätzlich werden die Metadaten für den Export des globalen

Schemas auch noch in die bei der Erstellung angegebene Datenbank (Dimension-Repository) exportiert.

Der Export der Metadaten in die Datenbank teilt sich in verschiedene Blöcke, welche den unterschiedlichen Tabellen des Metadata-Dictionaries zugeordnet sind. Die Unterteilung erfolgt z.B. nach der Erstellung von Würfeln oder der Erstellung von Dimensionen. Dies soll den Effekt haben, dass etwaige Fehler beim Export einfacher lokalisiert werden können. Für jeden abzuarbeitenden Block wird separat eine Datenbankverbindung aufgebaut, somit können eventuelle Syntaxfehler den verschiedenen Blöcken zugeordnet werden.

Der beschriebene Prozess wird in der Klasse `ExportMetaDataToDatabase` ausgeführt. Der Methode `create()` wird der Pfad zum Container für die UML-Modelle übergeben, wodurch in weiterer Folge die entsprechenden Schemata geladen werden. Durch die Methode `createSqlMetaDataExpressions()` werden die verschiedenen Schemata ausgelesen und die darin enthaltenen Elemente traversiert. Während der Traversierung der Modelle werden die erzeugten SQL-MDi Ausdrücke in Tabellen, eine für jeden Block, zwischengespeichert. Nachdem alle Schemata geparst wurden, können die einzelnen Blöcke in die Datenbank geschrieben werden.

In der Methode `create()` werden zusätzlich relevante Information für das Dimension-Repository exportiert. Als Zieldatenbank wird dazu jene Datenbank verwendet, welche beim Erstellen des Global-Schema angegeben wurde. Die Daten werden um die Kompatibilität mit dem SQL-MDi Query Processor aufrecht zu erhalten in einer Oracle Datenbank gespeichert. Durch den Exportprozess werden die leeren Fakt- und Dimensionstabellen erzeugt, welche die Grundlage für die Auswertungen des SQL-MDi Query Processors bereitstellen.

7.3.2 Komponente SQL-MDi Datei erzeugen

Zum Erzeugen der SQL-MDi Datei werden alle Mapping-Dateien (Namenskonvention: **.map* und **.gmap*) eingelesen und die darin enthaltenen SQL-MDi Ausdrücke zusammengeführt. Es muss die passende Reihenfolge eingehalten werden. Aus diesem Grund müssen die Ausdrücke der jeweiligen Import-Schemas vor dem Global-Schema durchgeführt werden. Abbildung 6-13 veranschaulicht diesen Prozess.

Beim Export sowie beim Erzeugen der Vorschau im Mapping-Editor werden die enthaltenen Platzhalter durch den SQL-MDi Ausdruck des passenden Elements ersetzt. Beispielsweise könnte bei der Erstellung eines Import-Mappings für einen Würfel folgender Ausdruck entstehen.

```
CUBE dw2::verkauf AS c2
 (MEASURE  c2.umsatz,  Dim  c2.datum   (MAP  LEVELS  c2.datum  ([datum],  [month  ->  monat],
[jahr])))
```

Listing 7-8: SQL-MDi einfaches Beispiel zu Cube-Mapping

Nachdem alle SQL-Ausdrücke erstellt wurden, können diese durch den von [Brunneder 2008] entwickelten SQL-MDi Query Parser validiert werden. Da dieser Parser nur eine Datei und keinen String als Eingabewert akzeptiert, muss die Datei bereits vor der Validierung gespeichert werden. Dies hat den Nachteil, dass etwaige Fehler erst nach dem Erstellen der Datei bekannt werden. Sollten Fehler auftreten, wird dem Benutzer eine Pop-up Nachricht angezeigt und eine *.log*-Datei erzeugt. Um eine Ausgabe der Fehler in einer separaten Datei zu ermöglichen, waren einige marginale Änderungen an dem SQL-MDi Query Parser notwendig. Im SQL-MDi Query Parser werden die bei der Validierung erkannten Fehler in der Konsole ausgegeben, für den Global Schema Architect war das jedoch nicht ausreichend. Somit wurden Variablen zur Zwischenspeicherung der jeweiligen Fehlermeldungen verwendet, welche im Nachhinein in die *.log*-Datei übernommen werden. Damit diese Änderungen durchgeführt werden können, wurde die Standalone-Version (jene ohne grafische Benutzerschnittstelle) direkt eingebunden und befindet sich im Paket `at.thesis`.

Um den finalen Ausdruck wie in Listing 7-8 dargestellt zu erhalten, nimmt der Global Schema Architect einige Ersetzungen vor. Listing 7-9 beschreibt die einzelnen in diesem Zusammenhang bedeutenden Bestandteile. Die Unterteilung der SQL-MDi Ausdrücke gliedert sich nach den Ebenen des Metamodells (siehe Abbildung 5-5). Deshalb befindet sich der Ausdruck des Würfels (*Cube*) auf der *Schemaebene*. Ausdrücke von Dimension und Kenngrößen (*Measures*) befinden sich auf der *Würfelebene*. Der genaue Ausdruck zu den einzelnen Levels ist schlussendlich in der *Dimensionsebene* definiert.

```
//Cube - Schemaebene
CUBE dw2::verkauf AS c2
(MEASURE <measure#umsatz#>, Dim <dim#datum#>)

//Dimension Datum - Würfelebene
c2.datum  (MAP LEVELS c2.datum ([<lev#datum#>], [<lev#month#>], [<lev#jahr#>]))

//Measure umsatz - Würfelebene
c2.umsatz

//Level datum - Dimensionsebene
datum

//Level month - Dimensionsebene
month -> monat

//Level jahr - Dimensionsebene
jahr
```

Listing 7-9: Bestandteile eines SQL-MDi Ausdrucks

Die Umsetzung dieser Komponente erfolgt in der `ExportMappingFile` Klasse. Die Ersetzung der jeweiligen Elemente erfolgt durch die private Methode `getContents()` welche von der Methode `getSqlMdiExpressions()` aufgerufen wird. Im Zuge dessen werden die notwendigen Ersetzungen durch reguläre Ausdrücke vorgenommen. In Listing 7-9 sind die einzelnen Bestandteile strukturiert dargestellt. So wird z.B. der Platzhalter `<lev#month#>` durch den Ausdruck `month -> monat` ersetzt.

7.4 Integration des Plug-ins in die Eclipse Plattform

Eine genaue Beschreibung wie die Integration eines Plug-ins an die Eclipse-Plattform funktioniert wurde bereits in Abschnitt 4 ausführlich erläutert. Dieser Abschnitt beschäftigt sich daher mit der konkreten Umsetzung für den Global Schema Architect.

Ein zentraler Punkt in diesem Zusammenhang ist die Anbindung des Profils an das Plug-in. Es müssen alle Pfade richtig gesetzt werden, um das Profil in der ausführbaren Version von Eclipse verwenden zu können. Die Anbindung erfolgt durch den Einsatz eines Extension Points, welcher den Pfad für die benötigte Datei anzeigt.

```
<extension point="org.eclipse.uml2.uml.generated_package">
  <profile
     uri="http://www.eclipse.org/uml/gsa"
location="pathmap://MYGSA_PROFILES/Gsa.profile.uml#_M2IecOGUEdyvwJLpBaSlhQ">
  </profile>
</extension>
```

Listing 7-10: Anbindung des Profils

Der in Listing 7-10 illustrierte Code befindet sich in der Datei *plugin.xml*. Im Tag `profile` befindet sich der Pfad zum obersten Element des UML-Profils. Der Ausdruck `MYGSA_PROFILES` repräsentiert den Pfad `platform:/plugin/org.eclipse.uml.gsa/model/`, der zuvor als Extension zwischengespeichert wurde.

Neben dem Profil werden in der *plugin.xml* Datei auch alle *Wizards* des Werkzeuges angegeben um von Eclipse aufgerufen werden zu können. Sollte ein Element nicht in dieser Datei angefügt sein, wird es von der Plattform nicht erkannt.

Zusätzlich müssen die Dateien *MANIFEST.MF* und *build.properties* angelegt werden um die abhängigen Pakete zu definieren. So enthält letztere z.B. die Referenzen auf die für den Datenbankimport zu verwendenden Treiber *ojdbc14.jar* und *sqljdbc.jar*.

7.5 Zusammenfassung

Aufbauend auf der in Kapitel 6 erstellten Systemarchitektur wurde in diesem Kapitel die Implementierung des Werkzeuges beschrieben. Im Zuge der Beschreibung der Implementierung wurde die Umsetzung mancher Komponenten mithilfe von Quelltextfragmenten erläutert.

Die Systemimplementierung gliedert sich ebenso wie die Systemarchitektur in drei Komponentengruppen: schemabezogene Komponenten, Mapping-Komponenten und Export-Komponenten. In diesem Kapitel wurde zusätzlich noch auf die Erstellung der Datei *plugin.xml* eingegangen, da diese einen zentralen Charakter zur Einbindung in die Eclipse-Plattform hat.

8 Evaluierung und Integrationstest

In diesem Abschnitt wird im Speziellen auf die Evaluierung und das Zusammenspiel zwischen den verschiedenen Komponenten des föderierten DWs eingegangen. Die Evaluierung wird mittels eines Integrationstest durchgeführt.

Der Global Schema Architect stellt einen Teil eines föderierten DW-Systems dar. Um eine ordnungsgemäße Funktionalität des Werkzeugs zu gewährleisten, ist ein über die verschiedenen Komponenten des föderierten DW-Systems übergreifender Test notwendig (Integrationstest).

In einem föderierten DW-System übernimmt der Global Schema Architect die Bereitstellung der Daten für eine ordnungsgemäße Abwicklung des Abfrageprozesses, welcher durch den SQL-MDi Query Processor sowie durch den SQL-MDi Query Parser durchgeführt wird. Demnach wird der Erfolg des Integrationstest an der Kompatibilität mit den anderen Werkzeugen gemessen. Die Interaktion erfolgt über die vom Global Schema Architect bereitgestellten Schnittstellen Metadata-Dictionary, Dimension-Repository und einer SQL-MDi Datei, welche die Heterogenitäten zwischen den einzelnen Instanzen der Datamarts überbrückt.

8.1 Aufbau und Rahmenbedingungen des Integrationstests

Um die erfolgreiche Integration des Global Schema Architect in das föderierte DW System zu testen, mussten sowohl der SQL-MDi Query Processor als auch der SQL-MDi Query Parser herangezogen werden. Der Integrationstest orientierte sich an dem bereits von [Rossgatter 2008] für den SQL-MDi Query Parser durchgeführten Test.

Der Unterschied zum Test von [Rossgatter 2008] besteht in der Bereitstellung der Daten. So wurde dort das Metadata-Dictionary, das Dimension-Repository sowie die SQL-MDi Datei als gegeben angesehen. Während des Integrationstests für den Global Schema Architect wurden diese Elemente vom Benutzer mithilfe des Werkzeuges erstellt. Um die korrekte Funktionalität des Global Schema Architects sicher zu stellen, mussten dieselben Ergebnisse wie bei dem von [Rossgatterer 2008] durchgeführten Integrationstest erzielt werden.

Als Grundgerüst wurde das in Kapitel 1.2 vorgestellte Rahmenbeispiel herangezogen. Die Datenbanken der Unternehmen MFA und MFB wurden auf einem MS-SQL-Server als gegeben vorausgesetzt. Eine genaue Dokumentation der Datenbanken in Form von SQL-Ausdrücken findet sich auf der mit dieser Arbeit ausgelieferten CD im Ordner *db_scripts* in den Dateien *createDW_dw1.sql* und *createDW_dw2.sql*.

Durch Benutzerinteraktion wurden die importierten Schemata angepasst, das globale Schema erzeugt und die entsprechenden Mappings gesetzt. Dieser Prozess wird im Rahmen dieses Integrationstests nicht näher beschrieben. Eine genaue Beschreibung der möglichen Tätigkeiten kann im Benutzerhandbuch zu diesem Projekt nachgeschlagen werden. Am Ende dieses Vorgangs standen dem Benutzer zwei **.import.uml*, eine **.gsa.uml*, zwei **.map* und eine **.gmap* Datei zur Verfügung.

Das Metadata-Dictionary und das Dimension-Repository werden durch die Exportfunktionen „Export Metadata to Database" des Global Schema Architects bereitgestellt. Die Speicherung des Metadata-Dictionaries erfolgte ebenso wie die Speicherung der Schemata der Beispielunternehmen auf einer Datenbank des MS-SQL Servers (*OLAP_Repository*). Das Dimension-Repository wurde auf einer bei der Erstellung des Global-Schema erzeugten Instanz eines Oracle-Datenbankservers angelegt. Eine genaue Auflistung der zu exportierenden Metadaten findet sich wiederum auf der beigelegten CD in der Datei *fillMetadataDictionary.sql* im Ordner *db_scripts*.

Basierend auf den **.map* Dateien und der **.gmap* Datei wurde die endgültige SQL-MDi Datei für die Eingabe im SQL-MDi Query Processor erzeugt. Für eine korrekte Abarbeitung des Rahmenbeispiels musste die SQL-MDi Datei dem Listing 8-1 entsprechen.

```
DEFINE CUBE dw1::verkauf AS c1
 (MEASURE c1.dauer, MEASURE c1.umsatz_telefonie, MEASURE c1.umsatz_sonstiges, Dim c1.datum_std , Dim c1.svnr , Dim c1.mobilnetz ,
 PIVOT MEASURES c1.umsatz_telefonie, c1.umsatz_sonstiges INTO c1.umsatz USING c1.umsatzkategorie)
 (ROLLUP c1.datum_std TO LEVEL c1.datum[datum] )
 CUBE dw2::verkauf AS c2
 (MEASURE c2.gespraechsdauer -> dauer, MEASURE c2.umsatz, Dim c2.datum (MAP LEVELS c2.datum ([datum], [month -> monat], [jahr])), Dim c2.kunde -> svnr (MAP LEVELS c2.kunde ([kunde -> svnr], [tarif])), Dim c2.mobilnetz , Dim c2.umsatzkategorie )
 (CONVERT MEASURES APPLY usd2Eur() FOR c2.umsatz DEFAULT)
 GLOBAL CUBE MAISLINGER::verkauf AS c0
 MERGE DIMENSIONS c1.datum AS d1, c2.datum AS d2 INTO c0.datum AS d0
 MERGE DIMENSIONS c1.kunde AS d3, c2.kunde AS d4 INTO c0.kunde AS d5
(RELATE d3.svnr,d4.kunde WHERE d3.svnr=d4.kunde USING HIERARCHY OF d3)
(MATCH ATTRIBUTES d3.bezeichnung IS d4.name)
(CONVERT ATTRIBUTES APPLY usd2Eur() FOR d4.grundgebuehr DEFAULT)
 MERGE DIMENSIONS c1.mobilnetz AS d6, c2.mobilnetz AS d7 INTO c0.mobilnetz AS d8
(RENAME d6.mobilnetz >> 'HandyTelInc' WHERE c1.mobilnetz='HandyTel')
   MERGE DIMENSIONS c1.umsatzkategorie AS dp33, c2.umsatzkategorie AS d10 INTO c0.umsatzkategorie AS d11
 MERGE CUBES c1, c2 INTO c0 ON datum, svnr, mobilnetz, umsatzkategorie
AGGREGATE MEASURE umsatz IS SUM OF umsatz,
AGGREGATE MEASURE dauer IS SUM OF dauer
(MEASURE umsatz,MEASURE dauer, DIM datum, DIM svnr, DIM mobilnetz, DIM umsatzkategorie)
```

Listing 8-1: Exportierte SQL-MDi Datei

Die explizite Beschreibung der Konfliktlösungen in Listing 8-1 können in [Rossgatterer 2008 S. 151ff] nachgelesen werden. In diesem Test wurde die syntaktische und semantische Korrektheit der SQL-MDi Ausdrücke vorausgesetzt.

8.2 Ergebnis des Integrationstests

Um die Korrektheit der von dem Global Schema Architect bereitgestellten Schnittstellen zu überprüfen, wurde die SQL-MDi Datei in Verbindung mit der in Listing 8-2 dargestellten OLAP Abfrage im SQL-MDi Query Parser verarbeitet.

```
SELECT d.jahr, d.monat, k.svnr AS kunde, m.mobilnetz, SUM(v.umsatz) AS umsatz
FROM datum d, mobilnetz m, verkauf v, kunde k
WHERE v.svnr=k.svnr AND v.datum=d.datum AND m.mobilnetz=v.mobilnetz
GROUP BY ROLLUP (d.jahr, d.monat, k.svnr, m.mobilnetz);
```

Listing 8-2: OLAP Abfrage für den Integrationstest

Das Ergebnis dieser Abfrage im SQL-MDi Query Parser auf Basis der durch den Global Schema Architect bereitgestellten Daten lieferte die folgende Tabelle.

Jahr	Monat	Kunde	Mobilnetz	Umsatz
2006	1	1234010180	MobCom	26.87
2006	1	1234010180	HandyTelInc	7.72
2006	1	1234010180		34.59
2006	1	4567040871	FiveCom	28.60
2006	1	4567040871		28.60
2006	1			63.20
2006	6	9876543210	FiveCom	7.90
2006	6	9876543210	HandyTelInc	19.12
2006	6	9876543210		27.02
2006	6			27.02
2006	8	6785041070	HandyTelInc	7.46
2006	8	6785041070		7.46
2006	8			7.47
2006				97.68

Tabelle 8-1: Ergebnis OLAP-Abfrage für Integrationstest

Da die Tabelle 8-1 dem Ergebnis des Integrationstest von [Rossgatter 2008] entspricht, wird der Integrationstest als erfolgreich angesehen. Ohne einen richtigen Aufbau der vom Global Schema Architect bereitgestellten Schnittstellen wäre kein ordnungsgemäßes Ergebnis erzielt worden. Der Global Schema Architect wurde somit erfolgreich in das föderierte DW System integriert.

9 Resümee und Ausblick

In diesem Kapitel werden die Ansprüche an die Diplomarbeit, sowie die zur Erfüllung überwundenen Schwierigkeiten nochmals zusammengefasst. Schlussendlich wird ein Ausblick auf eventuelle weiterführende Arbeiten am Global Schema Architect gegeben.

9.1 Resümee

Das Ziel dieser Arbeit war es eine Teilkomponente (den Global Schema Architect) eines föderierten DW Systems zu erstellen. Der Global Schema Architect musste als Werkzeug zur Integration von unterschiedlichen Data Marts und zur Erstellung eines globalen Schemas umgesetzt werden. Die vom Global Schema Architect bereitgestellten Daten dienen als Eingabewerte zur Ausführung des Abfrageprozesses des SQL-MDi Query Processors.

Die Schwierigkeit in der Entwicklung des Global Schema Architects bestand überwiegend darin, die notwendigen durch den SQL-MDi Query Processor und den SQL-MDi Query Parser geforderten Schnittstellen bereitzustellen. Darüberhinaus mussten diese auf einfache und benutzerfreundliche Art und Weise umgesetzt werden. Ein zusätzlicher Aufwand ergab sich aus der Forderung Eclipse als Entwicklungs- und Laufzeitumgebung zu verwenden.

Als spezielle Stärke des Global Schema Architects kann die zentrale Verwendung des Metamodells gesehen werden. Durch den Einsatz eines Metamodells basieren alle Editoren auf einem einheitlichen Grundgerüst, wodurch die Interoperabilität zwischen den unterschiedlichen Data Marts ermöglicht wird. Darüberhinaus vereinfacht es die Interaktionen zwischen internen Komponenten und externen Schnittstellen.

Die Unterscheidung der eingesetzten Dateitypen erfolgt durch Namenskonventionen. Der Einsatz dieser ermöglicht es dem Benutzer den Aufgabenbereich jeder Datei leicht zu erkennen. Für den Benutzer ist es dadurch einfacher den Überblick über ein Projekt zu behalten.

Durch den Einsatz von Eclipse war eine fundierte und zeitaufwändige Studie der zu verwendeten Technologien unumgänglich. Ebenso war auch eine Einarbeitung in den modellgetriebenen Ansatz zur Entwicklung von Softwareprodukten notwendig. Nach vollzogener Techno-

logieexploration stellte sich der Einsatz von Eclipse als großer Vorteil heraus. Durch die Implementierung mithilfe des EMF war es möglich aus Modellen Teile des Quellcodes zu generieren. Speziell in der frühen Entwicklungsphase konnten so etwaige Designfehler zeitgerecht behoben werden, indem der notwendige Quelltext nochmals neu generiert wurde.

Eclipse stellte sich für die Umsetzung des Plug-Ins als Rich-Client-Plattform als geeignete Basistechnologie heraus. So passen sich die verschiedenen Elemente des Editors der Darstellung des jeweiligen Betriebssystems an. Im Gegensatz zu anderen Java-Frameworks wie z.B. Swing lässt sich die Oberfläche so problemlos in das jeweilige System integrieren.

Ein Ziel dieser Arbeit war es eine benutzerfreundliche Oberfläche zur erstellen. Der Mapping-Editor wurde durch den Master-Detail Ansatz auf übersichtliche Art und Weise umgesetzt. Die Repräsentation der einzelnen Mappings erfolgt in tabellarischer Form. Eine rein grafische Lösung würde bei steigender Anzahl an Data Marts Komplexitätsprobleme in der Darstellung der verschiedenen Zusammenhänge hervorrufen. Durch eine inkrementelle Erstellung der Mappings wird dem Benutzer eine schrittweise Lösung der Probleme ermöglicht. Dies hat den Vorteil, dass der Benutzer das System in separate, leichter verständliche Teile zerlegen kann.

Im Zuge der Implementierung des Werkzeuges mussten jedoch in einigen Bereichen Einschränkungen akzeptiert werden. So wurde z.B. die Struktur des DWs auf die Form eines Star-Schemas begrenzt. Ein Import von Snowflake-Schemas ist demnach nicht möglich.

Die ordnungsgemäße Funktionalität des Global Schema Architects wurde durch den Testfall im Integrationstest überprüft.

9.2 Ausblick

In diesem Abschnitt werden Erweiterungsmöglichkeiten für den Global Schema Architect vorgeschlagen um Ansätze für eventuelle weitere Entwicklungen zu geben.

Der Global Schema Architect kann in einigen Stellen erweitert werden. So könnten z.B. die Heuristiken für das Auslesen der Data Marts aus den verschiedenen Datenbanken spezieller auf unterschiedliche Datenbanktechnologien abgestimmt werden. Diese Arbeit beschränkt sich auf die Datenbanken MS-SQL und Oracle.

In der Bereitstellung der Vorschläge für das Erstellen der Mappings ist ebenso noch Verbesserungspotential vorhanden. So könnten die Vorschläge beispielsweise automatische Rückschlüsse aus der Struktur des globalen Schemas im Vergleich zum importierten Schema schließen. Für die Umsetzung dieses Ansatzes müsste die „Intelligenz" des Werkzeuges verbessert werden.

Der Global Schema Architect ist immer direkt vom SQL-MDi Query Processor abhängig, da die Erstellung der Mappings im Global Schema Architect mit der Verarbeitung der SQL-MDi Datei im SQL-MDi Quer Processor kompatibel sein muss. Aus diesem Grund wäre eine ganzheitliche Lösung dieses Umstands empfehlenswert. Ein Integration des SQL-MDi Query Processors in die Eclipse-Plattform würde zum einen die Interaktion zwischen den beiden Elementen des föderierten DW erleichtern und zum anderen ein einheitliches „*look and feel*" der Benutzeroberfläche erzeugen.

10 Anhang A

10.1 Benutzerhandbuch

In diesem Handbuch werden die Installation und das Einrichten des Systems beschrieben. Dazu werden die mit dem Handbuch ausgelieferten Dateien herangezogen (siehe beiliegende CD). Infolgedessen wird auf die durch den Global Schema Architect ermöglichten Handlungen und Anwendungsfälle eingegangen.

Das Handbuch bezieht sich auf das in Kapitel 1.2 beschriebene Rahmenbeispiel. Durch die Erläuterung der Anwendungsfälle anhand eines Beispiels soll die Verständlichkeit erhöht werden.

10.1.1 Installation

Um eine erfolgreiche Installation für das Rahmenbeispiel durchzuführen, sind gewisse Voraussetzungen zu erfüllen. Als Mindestanforderungen gelten Java 1.6, MS SQL-Server 2005 sowie Oracle Server. Zusätzlich muss am Oracle Server eine Instanz für das Dimension-Repository vorhanden sein. Nachfolgend werden die notwendigen Schritte zur Installation des Global Schema Architects angeführt. Die Skripten für die Anwendung in der Datenbank befinden sich im Order *db_scripts* der Distribution des Global Schema Architects.

1. Stellen Sie sicher, dass zumindest die Java Version 1.6 auf Ihrem System installiert ist.
2. Kopieren Sie den Ordner Eclipse von der CD in einen beliebigen Ordner auf Ihrem System. Für die Anwendung des sich auf der CD befindenden Programms, muss Windows als Betriebssystem eingesetzt werden. Die Plug-ins können auch in Eclipse-Versionen anderer Systeme angewendet werden. In diesem Ordner ist Eclipse samt allen Plug-ins enthalten, dadurch ergibt sich der hohe Platzbedarf von über 500 Megabyte.
3. Erstellen Sie die Data Warehouses für die zwei Mobilfunkunternehmen. Dazu legen sie im MS- SQL Server 2005 die Datenbanken *dw1* und *dw2* an.
4. Führen Sie daraufhin die Dateien *createDW_dw1.sql* bzw. *createDW_dw2.sql* in den entsprechenden Datenbanken aus.

5. Befüllen Sie die Datenbanken *dw1* und *dw2* mittels der Datenbankskripten *insertDW_Dw1.sql* und *insertDW_dw2.sql*.
6. Erstellen Sie die Datenbank zum Speichern der Metadaten. Hierzu legen Sie eine Datenbank mit dem Namen *OLAP_Repository* an.
7. Zum Erstellen der Struktur führen Sie das *createMetadataDictionary.sql* Skript in der *OLAP_Repository* Datenbank aus.
8. Stellen Sie sicher, dass Sie Zugang auf eine Instanz des Oracle Servers haben. In diesem Beispiel heißt die Instanz *maislinger*. Dies kann jedoch je nach Einsatz bzw. Instanz der Oracle Datenbank divergieren.
9. Weisen Sie beim ersten Start von Eclipse dem Projekt einen beliebigen Workspace zu.

Durch Ausführung der angeführten Schritte haben Sie die Grundlagen für die Anwendung des Projektes gelegt.

Mit dem Global Schema Architect wird die fertige Umsetzung des Rahmenbeispiels geliefert. Um dieses Beispiel zu laden, setzen Sie den Workspace beim Starten von Eclipse auf den Ordner *Workspace* Ihrer Distribution.

10.1.2 Anwendung

Der Global Schema Architect bietet eine Vielzahl von Möglichkeiten, um mit dem Werkzeug zu interagieren. Dieses Kapitel gliedert sich in die vom Benutzer ausführbaren Anwendungsfälle:

- Erzeugen eines neuen globalen Schemas,
- Importieren eines Data Marts,
- Erstellen der Import-Mappings,
- Erstellen des Global-Mappings,
- Export der SQL-MDi Datei und
- Export der Metadaten.

Die Beschreibung der einzelnen Anwendungsfälle wird jeweils mit einem oder mehreren Screenshots bildlich unterstützt. Zusätzlich wird bei hoher Komplexität des Anwendungsfalles ein UML-Aktivitätsdiagramm bereitgestellt.

Es wird vorausgesetzt, dass der Benutzer zumindest über ein rudimentäres Wissen über Eclipse verfügt. Die Beschreibung der Anwendungsdetails von Eclipse würde den Rahmen dieses Benutzerhandbuchs überziehen.

Zusätzlich wird ein Grundwissen über die multidimensionale Abfragesprache SQL-MDi als gegeben angesehen. Eine detaillierte Beschreibung der Abfragesprache findet sich in [Berger & Schrefl 2006]

Grundlage für jeden Anwendungsfall ist ein Eclipse-Projekt, welches zu Beginn erstellt werden muss. Hierzu muss in der Menüleiste *File → New → Project* ausgewählt werden. Daraufhin öffnet sich ein Assistent, welcher die verfügbaren Projekt- und Dateitypen anzeigt. In diesem Menü muss *General → Project* ausgewählt werden. Nach einem Klick auf die *Next*-Schaltfläche kann der Projektname vergeben werden. In diesem Fall wird der Name *user_manual* vergeben. Durch Betätigung der *Finish*-Schaltfläche wird das Projekt erstellt.

10.1.2.1 Erzeugen eines neuen globalen Schemas

Die Erstellung eines neuen Schemas kann in zwei unterschiedlichen Varianten durchgeführt werden. So gibt es zum einen die Möglichkeit, die Datei von Grund auf neu zu erstellen und zum anderen die Variante, ein vorhandenes importiertes Schema zu kopieren und in diesem entsprechende Änderungen vorzunehmen. Um zu dieser Auswahl zu gelangen, muss das Grundprojekt mit der rechten Maustaste angeklickt werden und im Kontextmenü *new → other → Global Schema Architect* ausgewählt werden. Innerhalb dieses Menüs kann der Benutzer zwischen den jeweiligen Varianten wählen. In Bezug auf die Umsetzung des Rahmenbeispiels wird ein neuer *Global Schema Architect File* erzeugt. Bei der Erstellung der Datei muss eine Datenbankverbindung zu einer Oracle-Datenbank angegeben werden, um dort das Dimension-Repository zu speichern. Zusätzlich muss ein passender Dateiname vergeben werden. Weiters besteht die Möglichkeit die Verbindung zur Datenbank zu testen. Der beschriebene Prozess wird in Abbildung 10-1 dargestellt.

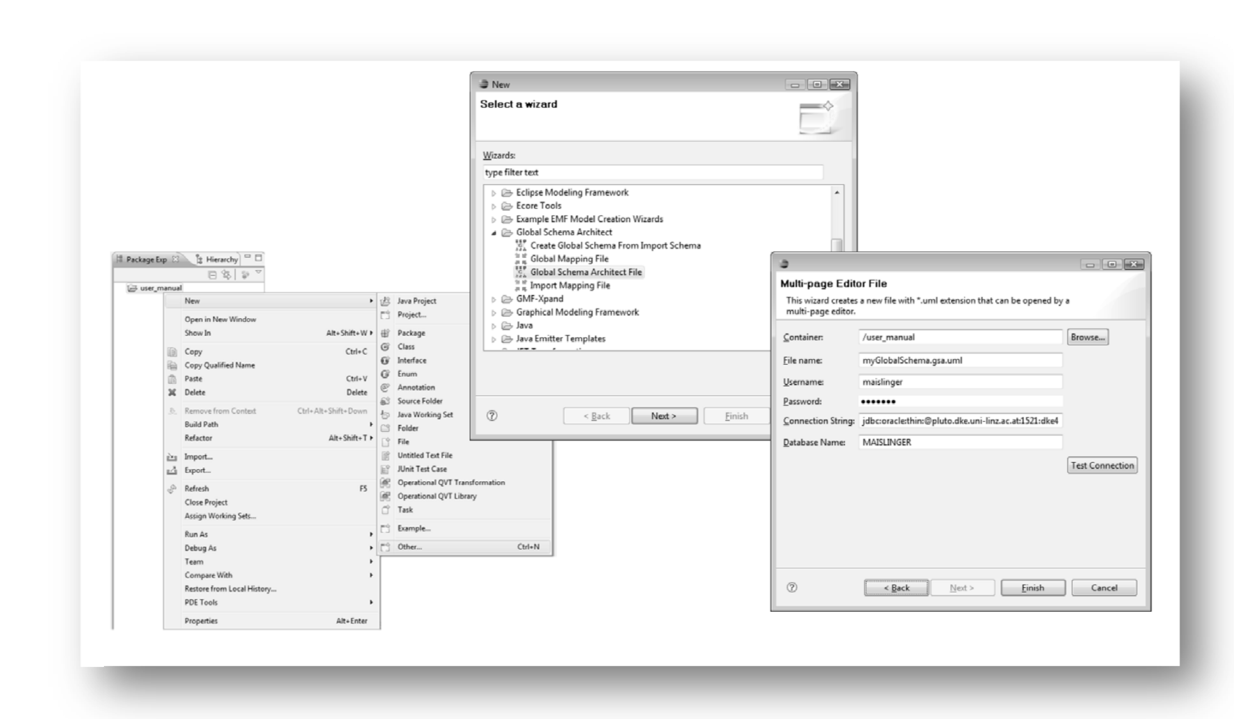


Abbildung 10-1: Prozess zur Erstellung des Global Schemas

Soll das globale Schema aus einem bereits vorhandenen Import-Schema erstellt werden, so wird dazu der *Create Global Schema From Import Schema* Assistent eingesetzt. Der Prozess verhält sich analog zum Erstellen einer leeren Datei, mit dem Unterschied, dass zusätzlich das zu kopierende Import-Schema angeführt werden muss.

Die erstellte Datei findet sich nun im Projekt und hat die Endung *gsa.uml*. Es besteht die Möglichkeit diese Datei mit einem grafischen Editor in Form eines Klassendiagramms zu bearbeiten. Dies erfolgt mit dem Befehl *Initialize Class Diagramm*, welcher im Kontextmenü der *.uml Datei* aufgerufen werden kann. Durch diesen Aufruf wird in weiterer Folge eine *.umlclass* Datei erzeugt, welche die grafische Repräsentation des UML-Modells darstellt. Abbildung 10-2 zeigt das Kontextmenü der UML-Datei sowie den Editor zum Bearbeiten der Dateien.

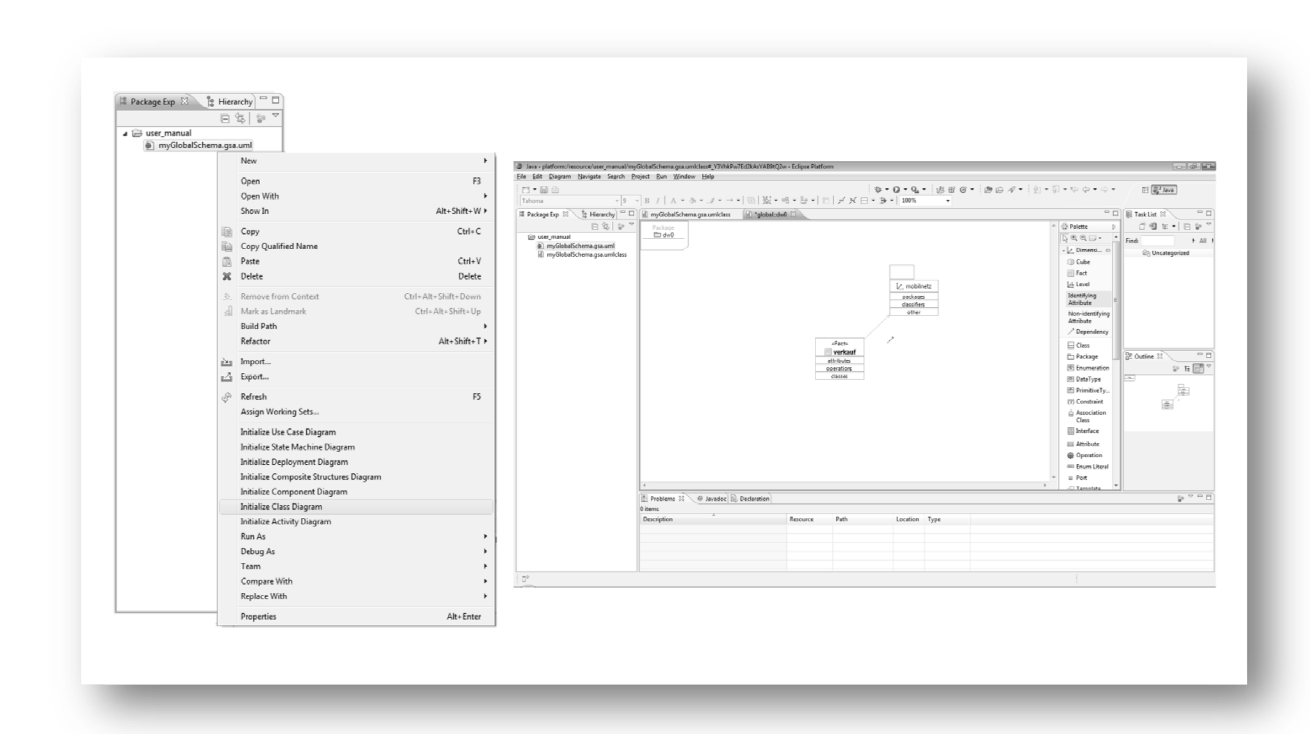


Abbildung 10-2: Editor für UML-Modelle

Die Bedienung des grafischen Editors ist einfach und selbsterklärend aufgebaut. Demnach lassen sich die Anwendungsmöglichkeiten ohne explizite Beschreibung erkennen und ausführen. Neue Elemente können im Editor durch einfache Klick-Anweisungen erstellt werden. Hierzu muss im Menü auf der rechten Seite, der sogenannten Palette, ein Element angeklickt werden. Danach muss im Zeichenbereich des Editors an der Stelle wo das Element abgelegt werden soll, ein weiterer Klick gesetzt werden.

Pakete (Dimension und Würfel) ermöglichen es, durch Doppelklick auf das oberste Quadrat der grafischen Darstellung, deren Inhalt anzuzeigen. Daraufhin wird der Inhalt des Paketes in einem neuen, separaten Fenster dargestellt. Zwischen den verschiedenen Fenstern kann daraufhin in den Registerkarten von Eclipse navigiert werden. Aus diesem Umstand ergibt sich der Vorteil, dass die Anzahl der Elemente pro Ansicht begrenzt bleibt, weil diese auf mehrere Pakete verteilt werden können. Ein Umbenennen eines Elements erfolgt mittels eines einfachen Klicks auf dessen Namen.

Weitere Einstellungen können in der *Properties*-Ansicht gesetzt werden, so kann dort z.B. der Dateityp eines Attributs festgelegt werden. Die *Properties* -Ansicht wird in Form eines zusätzlichen Fensters in der Eclipse Entwicklungsumgebung angezeigt.

Im Kontextmenü eines Elements können zusätzlich die passenden Stereotypen hinzugefügt werden. Der Editor ermöglicht ausschließlich ein Hinzufügen von Stereotypen. Das Löschen von Stereotypen ist in diesem Editor nicht möglich. Modellelementen können durch die *Delete* Schaltfläche im Kontextmenü durchgeführt werden.

Abbildung 10-3 stellt die beiden besprochenen Tätigkeiten dar. Im unteren Bereich der Abbildung sieht man das *Properties*-Menü des Fakts *verkauf*. Im Kontextmenü werden die zur Auswahl stehenden Stereotypen unter dem Menüpunkt *Apply Stereotype* angezeigt.

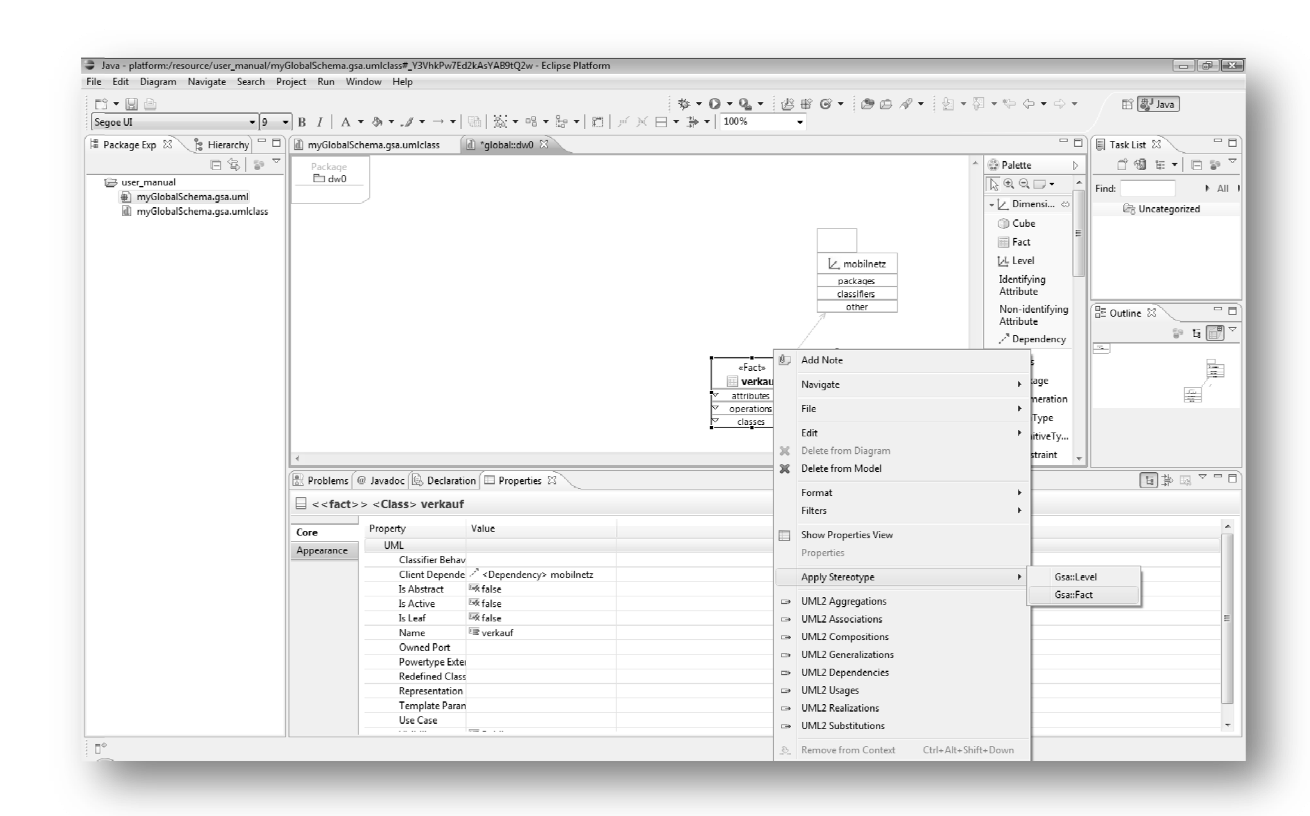


Abbildung 10-3: Properties und Hinzufügen von Stereotyp in grafischen Editor

Es besteht weiters die Möglichkeit das globale Schema auf richtigen Aufbau zu prüfen. Um eine Prüfung durchzuführen, muss der Editor für die UML-Datei (nicht der für den grafischen Editor) geöffnet werden. Dies erfolgt durch einen Doppelklick auf die **.uml* Datei. Der Editor der *.uml* Datei stellt die Struktur der Datei in Form eines Baumes dar. Um eine Validierung durchzuführen, muss in der Baumstruktur ein Würfelelement mit dem Stereotyp *<<cube>>* markiert werden und in der Menüleiste *UML Editor → Validate* ausgewählt werden. In Abbildung 10-4 wird der Aufruf der Validierung sowie das entsprechende Resultat angezeigt.

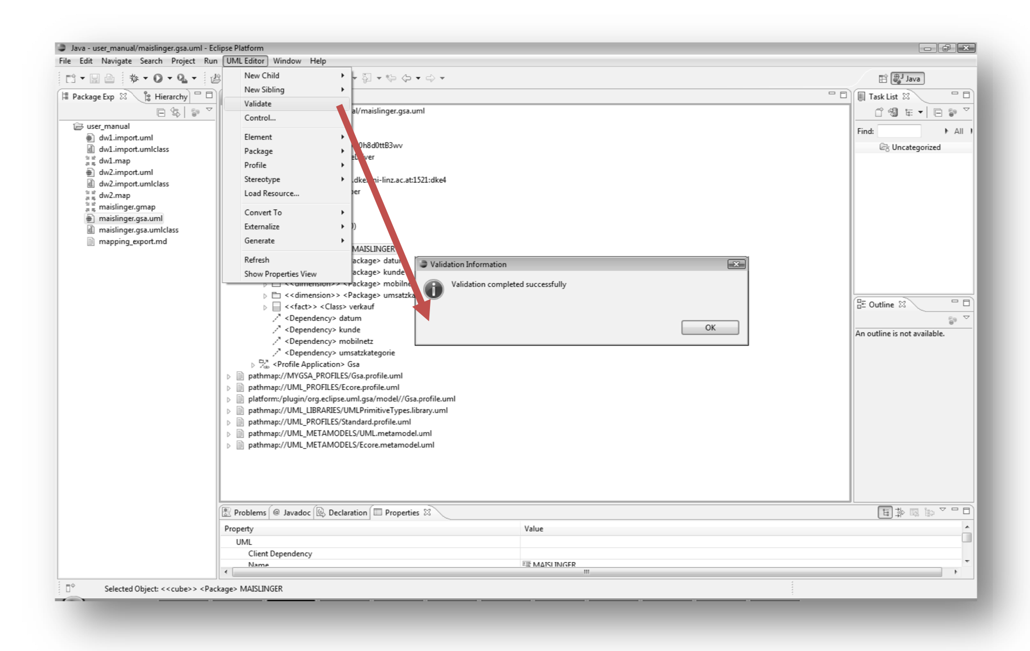


Abbildung 10-4: Validierung von Global Schema

Für die Umsetzung des Rahmenbeispiels, muss die Struktur des Global-Schemas wie in Abbildung 10-5 dargestellt aufgebaut sein.

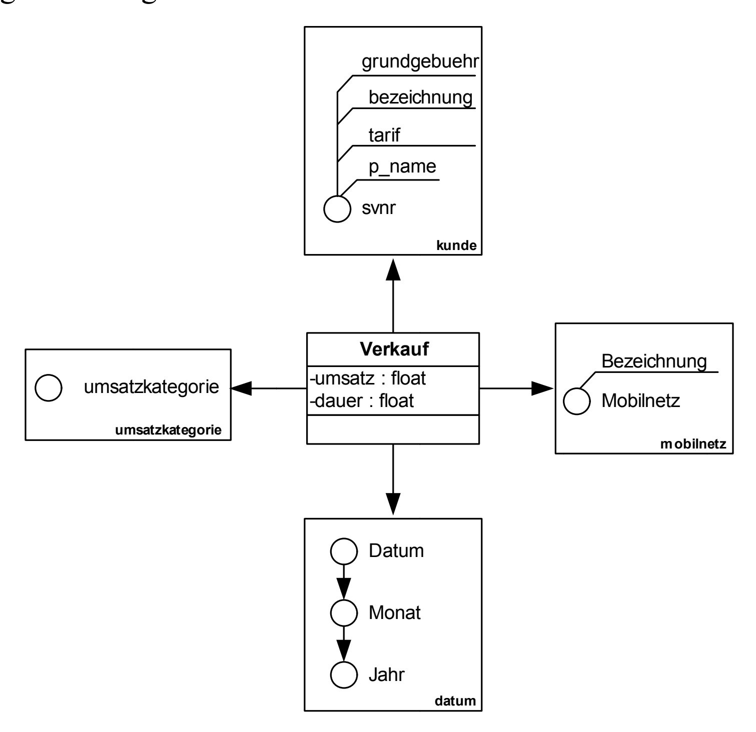


Abbildung 10-5: Aufbau des Global-Schemas

10.1.2.2 Import eines Data Marts

Um einen Data Mart zu importieren, muss eine Verbindung zur Datenbank aufgebaut werden. Daraufhin wird vom System ein Vorschlag für das zu importierende Schema gegeben, der optional durch den Benutzer angepasst werden kann. Nach Import des Schemas können die genauen Eigenschaften des Data Marts bearbeitet werden. Hierzu wird wiederum der in 10.1.2.1 beschriebene grafische Editor verwendet.

Die Grundidee des Importprozesses wird in Abbildung 10-6 dargestellt. Man erkennt, dass durch den Import nur eine bedingte Genauigkeit erreicht werden kann. So können z.B. von einem Star-Schema keine Hierarchien ausgelesen werden. Die Hierarchien müssen im nächsten Schritt durch den grafischen Editor vom Benutzer hinzugefügt werden.

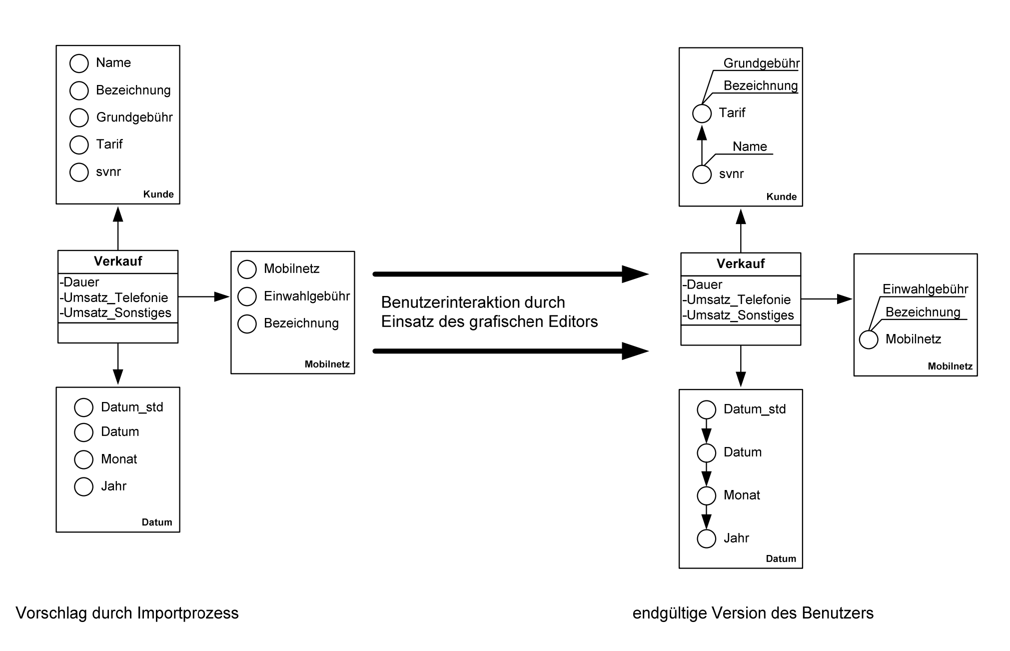


Abbildung 10-6: Importprozess

Die durchzuführenden Benutzerinteraktionen zum Erlangen des Vorschlages des Importprozesses werden im folgenden UML-Aktivitätsdiagramm dargestellt. Dabei werden die Aktivitäten des Benutzers sowie die Zustände des Systems angezeigt. In dieser Abbildung wird die Option des nachträglichen Ausbesserns des Vorschlages explizit hervorgehoben.

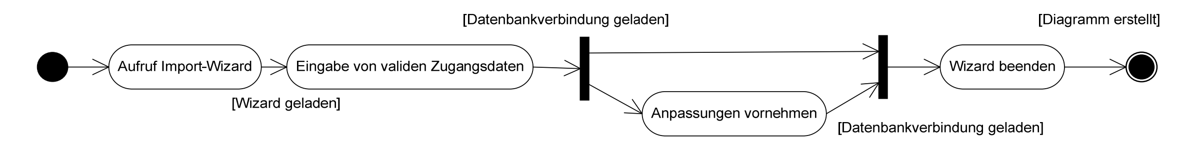
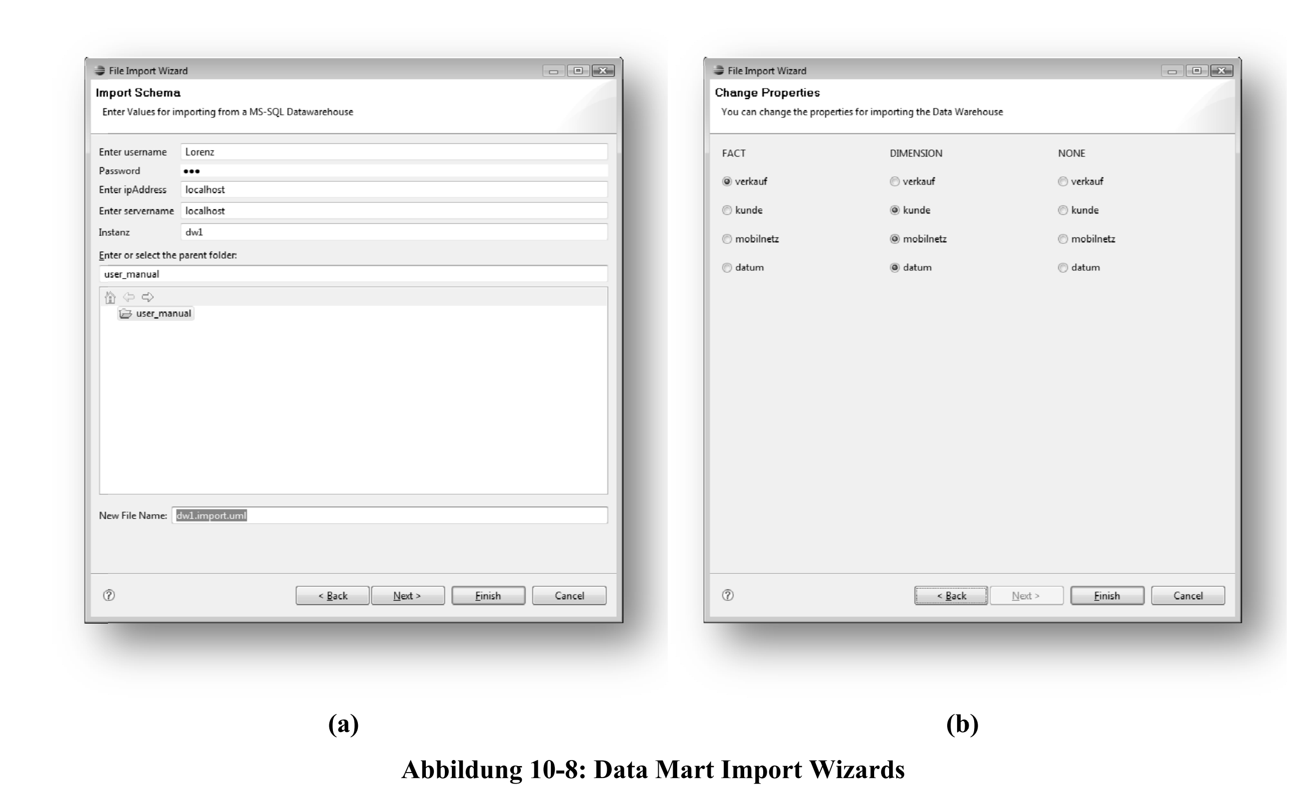


Abbildung 10-7: Aktivitätsdiagramm des Importprozesses

Zur Durchführung des Importprozesses wird der *Import Wizard* eingesetzt. Dieser kann im Kontextmenü des Projekts unter *Import* im Ordner *Global Schema Editor* ausgeführt werden. Dort kann zwischen einem Import aus einer MSSQL oder Oracle Datenbank gewählt werden. In Bezug auf das Rahmenbeispiel ist der *MSSQL Import* auszuwählen.

Nach Auswahl des *MSSQL Imports* erscheint der in Abbildung 10-8a dargestellte Assistent, in welchem die Zugangsdaten zur Datenbank eingegeben werden. Die Verbindung wird automatisch nach Eingabe aller relevanten Daten validiert. Sollte keine Verbindung erstellt werden können, wird eine Fehlermeldung angezeigt. Wichtig ist, dass ab der ersten erfolgreichen Verbindung zur Datenbank keine weitere Verbindung erstellt werden kann. Um eine neue Verbindung zu erstellen, muss der Assistent neu gestartet werden. Bei erfolgreicher Verbindung zur Datenbank wird ein Schemavorschlag geladen. Es besteht die Möglichkeit, diesen mittels der *Next*-Schaltfläche einzusehen oder auf die Heuristiken des Systems zu vertrauen und die Schemadatei direkt ohne weitere Einsicht des Vorschlags zu erstellen.



(a) **(b)**

Abbildung 10-8: Data Mart Import Wizards

Die Bearbeitung des Vorschlags erfolgt durch den in Abbildung 10-8b dargestellten Assistenten. Es besteht die Möglichkeit, die Interpretation der Tabellen zu ändern. So kann ein Fakt mit geringem Aufwand in eine Dimension umgewandelt werden. Hierzu muss der Options-

Auswahl auf das auszuwählende Element gesetzt werden. Vergleicht man den gegebenen Vorschlag mit dem Rahmenbeispiel, ist zu sehen, dass der gegebene Vorschlag richtig ist und somit keine Anpassungen notwendig sind.

Die beim Import erstellte Datei muss der Namenskonvention **.import.uml* gerecht werden, um auch für weitere Auswertungen anwendbar zu sein. Während des Ladens und des Erstellens der Datei erscheint ein Statusbalken, welcher den aktuellen Fortschritt der Operation anzeigt.

10.1.2.3 Erstellen der Import-Mappings

Import-Mappings können Konflikte auf Schemaebene überwinden. Zwischen jedem Import-Schema und dem Global-Schema wird dazu ein eigenes Mapping erzeugt. Beim Erstellen der Import-Mappings wird inkrementell ein SQL-MDi Ausdruck erstellt. Dies bedeutet, dass der endgültige SQL-MDi Ausdruck Stück für Stück zusammengestellt wird.

Um eine Import-Mapping Datei anzulegen, muss im Kontextmenü des Projekts *new → other → Global Schema Architect → Import Mapping File* ausgewählt werden. In dem daraufhin erscheinenden *Wizard* muss der Benutzer ein Global- und ein Import-Schema angeben. Die Auswahl unterstützt der Wizard durch Anzeigen der im Projekt vorhandenen Schemata. Nach Abschluss des Assistenten wird die Mapping-Datei (**.map*) erzeugt.

Der Editor für Mapping-Dateien besteht aus drei Bereichen, welche durch die Registerkarten im linken unteren Eck ausgewählt werden können:

- **Schema Editor**: Dies ist der grundlegende Editor für eine Mapping-Datei. Hier können die SQL-MDi Ausdrücke erstellt werden.
- **Code**: Darstellung des Quelltextes der Datei.
- **Preview**: In dieser Ansicht wird der endgültig erstellte SQL-MDi Ausdruck des jeweiligen Mappings angezeigt. Diese Darstellung dient ausschließlich der Kontrolle.

Der Schema Editor teilt sich in zwei Bereiche. In der linken Hälfte werden alle Elemente des importierten Schemas aufgelistet. In der rechten Seite werden die dazugehörigen Details angezeigt. Diese Darstellung kann nach Belieben vertikal oder horizontal erfolgen. In diesem

Benutzerhandbuch wird im weiteren Verlauf die horizontale Darstellung herangezogen. Die Darstellung der einzelnen Elemente wird durch Symbole (siehe Tabelle 10-1) verdeutlicht.

Nachfolgende Abbildung 10-9 zeigt einen Import-Mapping Editor. In dieser Abbildung sind u.a. die bereits erwähnten Registerkarten ersichtlich. Weiters lässt sich die strukturelle Aufteilung des Editors erkennen.

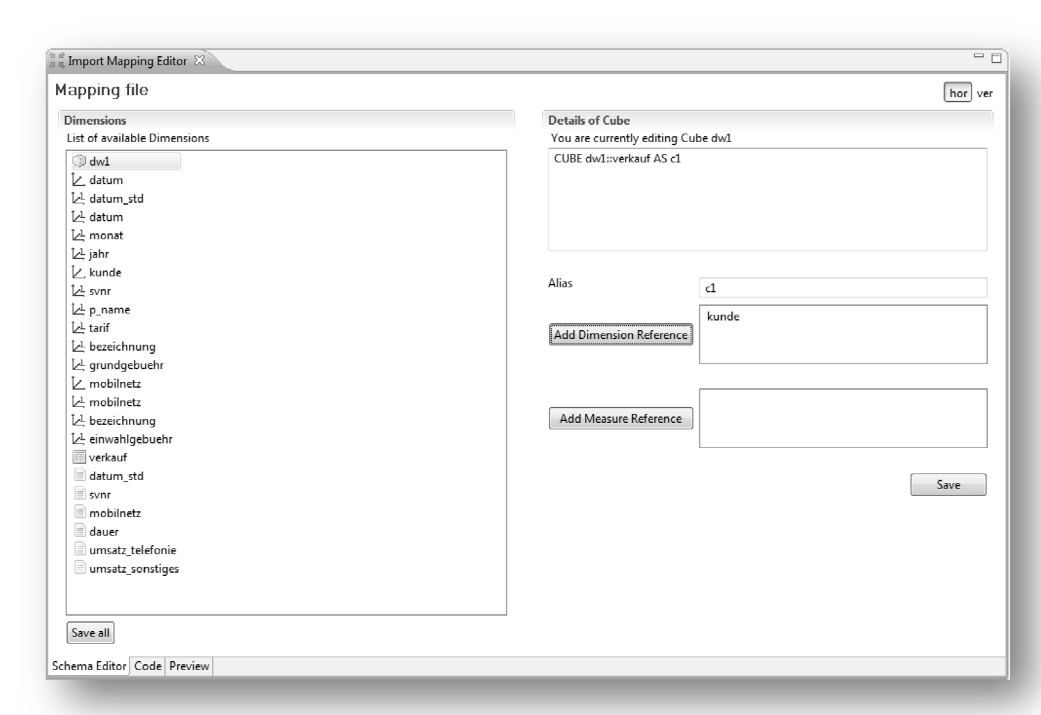


Abbildung 10-9: Darstellung Import-Mapping Editor

Die Darstellung der Detailseite divergiert nach den verschiedenen Elementen. So kann z.B. in der Detailseite zu einem Würfel (engl. Cubes) ein Alias für den Würfel angegeben werden. Zusätzlich können auch alle Dimensionen und Kenngrößen (engl. *Measures*) des Würfels, welche auf das globale Schema gemappt werden sollen, angeführt werden.

Generell gliedert sich die Detailseite von Würfel und Dimension in eine Schnellvorschau, sowie in die Erstellungsbereiche für die Mappings. Bei allen anderen Elementen wird keine Schnellvorschau bereitgestellt. Durch die in der Detailansicht angegebenen Schaltflächen hat der Benutzer die Möglichkeit, an Vorschläge des Global Schema Architects zu gelangen.
Abbildung 10-10 stellt in diesem Zusammenhang beispielsweise die Auswahl der Dimensionen in der Detailseite eines Würfels dar.

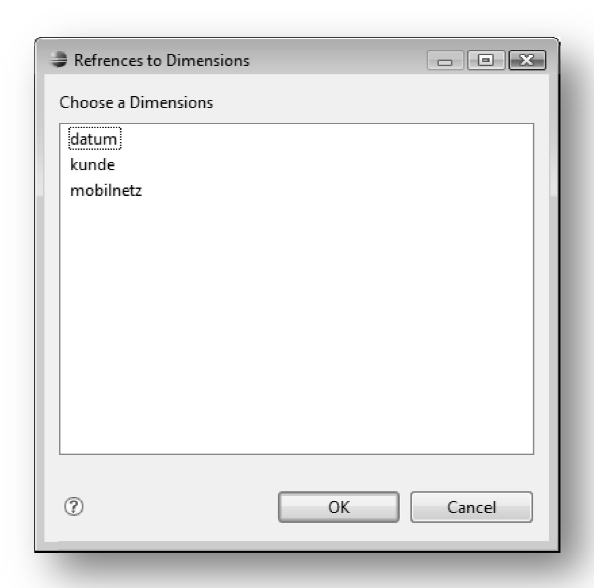


Abbildung 10-10: Dimensionsauswahl im Import-Mapping Editor

Die Information der Vorschläge wird aus verschiedenen Datenquellen erhoben. Je nach Anwendung wird das Global- oder das Import-Schema herangezogen. Um Heterogenitäten im Vergleich von Werten zu beseitigen, werden durch den Administrator Funktionen in Form einer XML Datei bereitgestellt. Die Verantwortung für die korrekte Definition der verschiedenen Methoden liegt beim Administrator. Eine Anpassung der Funktionen ist mit jedem neuen Release des Werkzeuges möglich. Der Funktionsumfang des Rahmenbeispiels beschränkt sich auf die Methode *usd2Eur()*.

Jede Detailseite enthält eine *Save*-Schaltfläche. Durch diese werden die durchgeführten Änderungen im jeweiligen Modell gesichert. Um Änderungen einer niedrigen Ebene, welche Auswirkungen auf die Schnellvorschau der übergeordneten Ebene haben, direkt in der Vorschau zu sehen, muss auch die übergeordnete Ebene gespeichert werden. Diese Problematik verdeutlicht beispielsweise das Hinzufügen eines Roll-up Ausdrucks zu einer Dimension. Diese Maßnahme hat eine Auswirkung auf die Schnellvorschau des Würfels und wird sofort in der Schnellvorschau angezeigt. Die Anzeige in der Vorschauansicht erfolgt erst nach dem Speichern der Schellvorschau. Folgende Abbildung stellt dieses Beispiel mittels eines UML-Aktivitätsdiagramms übersichtlich dar.

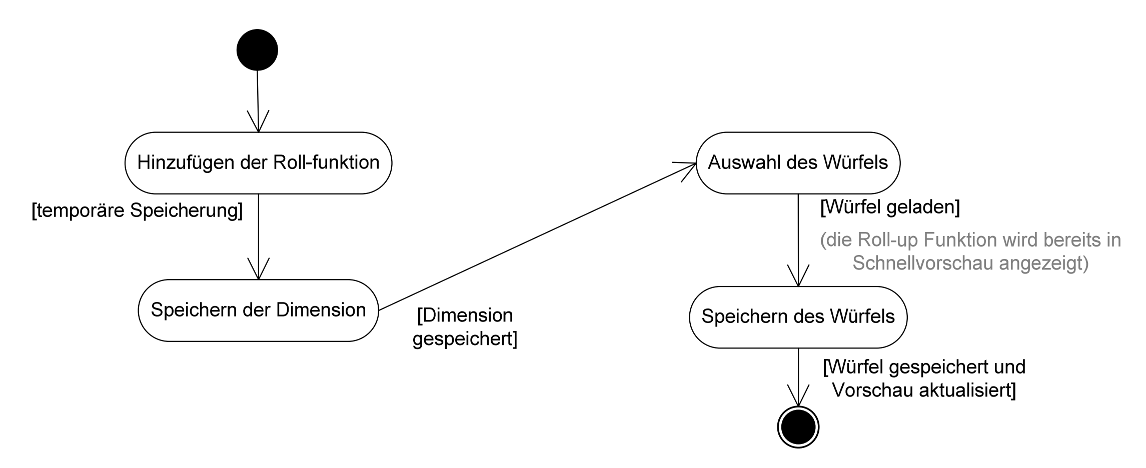


Abbildung 10-11: Aktivitätsdiagramm zur Speicherung eines Roll-up Ausdrucks

Zu den Import-Mappings muss angemerkt werden, dass keine Synchronisation mit dem Import-Schema oder dem Global-Schema erfolgt. Es wird immer der zum Zeitpunkt des Imports verwendete Stand herangezogen. Sollten nachträgliche Änderungen im Import-Schema notwendig sein, so muss auch die Mapping-Datei neu erzeugt werden.

10.1.2.4 Erstellen des Global-Mappings

Das Global-Mapping verhält sich in vielerlei Hinsicht sehr ähnlich dem Import-Mapping. Es sind jedoch auch einige signifikante Änderungen bzw. Unterschiede hervorzuheben. So werden durch das Global-Mapping Konflikte auf Instanz-Ebene gelöst. Daraus ergibt sich die Notwendigkeit für eine einzige Global-Mapping Datei.

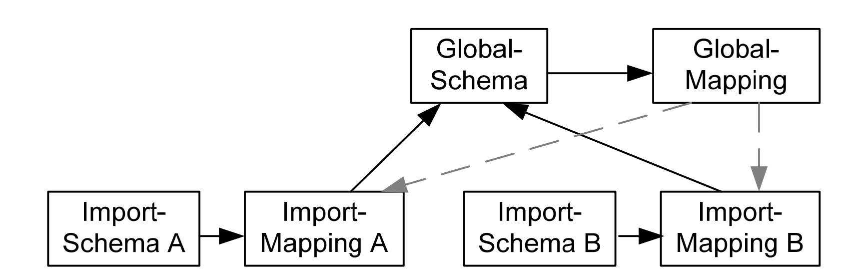


Abbildung 10-12: Aufteilung der Mappings

In Abbildung 10-12 wird die Aufteilung der Mappings dargestellt. In dieser Abbildung wird ersichtlich, dass nur ein Global-Mapping notwendig ist. Das Global-Mapping verlangt bei der Erstellung (*New → Other → Global Schema Architect → Global Mapping File*) nur die Angabe des Global-Schemas, die anderen notwendigen Elemente werden vom System dynamisch geladen. Für das Global-Mapping werden das Global-Schema sowie die Import-

Mappings herangezogen. Somit werden bereits überwundene Schemakonflikte bei der Erstellung des Global-Mappings berücksichtigt. Das Laden der Import-Mappings erfolgt somit dynamisch, wodurch bei Änderungen in den Import-Mappings das Global-Mapping nicht neu erstellt werden muss.

Ein weiterer großer Unterschied besteht darin, dass die Elemente der linken Seite des Editors nicht fest an ein Schema gebunden sind, weshalb beliebig viele Elemente erstellt werden können. Üblicherweise werdem pro Dimension des Global-Schemas eine *Merge Dimensions* und pro Cube eine *Merge Cubes* Anweisung hinzugefügt. Der einzige statische Eintrag der Tabelle ist *Cube Aliases*, womit an einer Stelle alle notwendigen Aliases (Dimensionen und Würfel) für ein Schema gesetzt werden können.

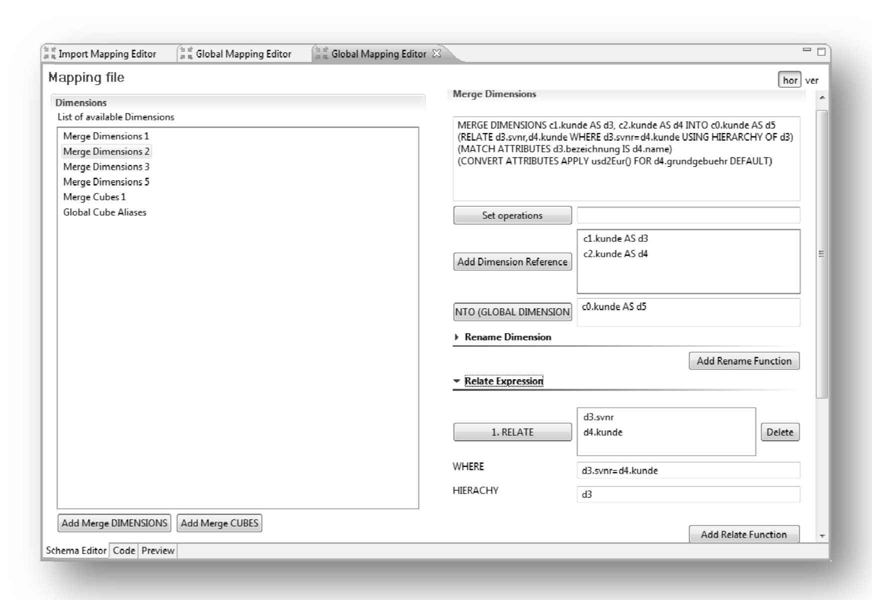


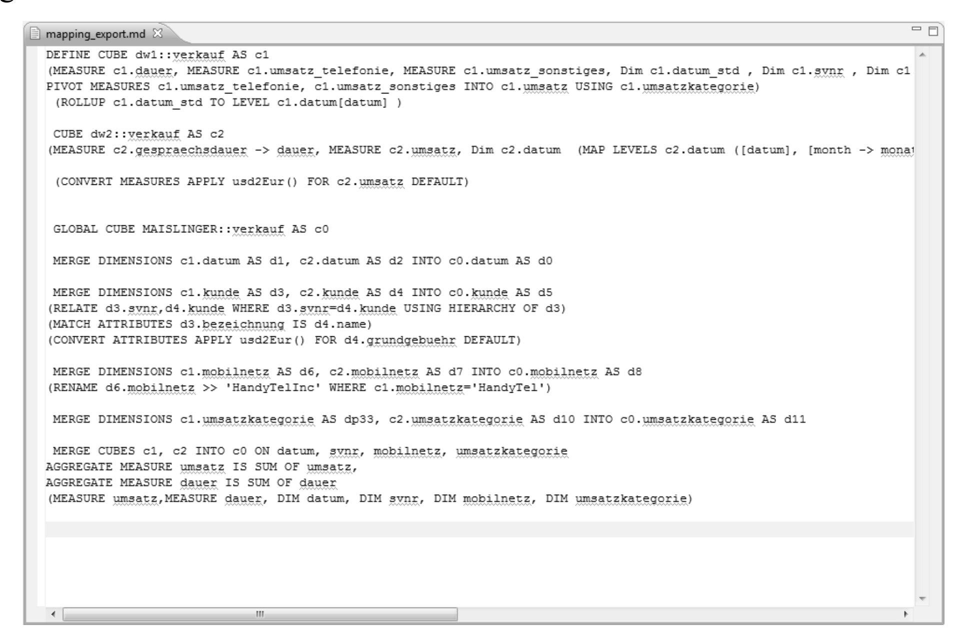
Abbildung 10-13: Global-Mapping Editor

In Abbildung 10-13 wird der Global-Mapping Editor dargestellt. Auf der linken Seite des Editors sind die hinzugefügten Elemente dargestellt. Der Aufbau der rechten Seite gleicht wiederum dem Import-Mapping. Die Ansicht des Global-Mappings unterscheidet sich zum Import-Mapping durch die Verwendung von Drop-Down Feldern. Durch diese kann einerseits Inhalt dynamisch hinzugefügt werden, und andererseits ist so eine übersichtliche und platzsparende Gestaltung auch bei einer Vielzahl an Ausdrücken möglich. So kann eine *Merge Dimensions* Anweisung z.B. mehrere *Relate* Ausdrücke enthalten. In den Import-Mappings ist im Gegensatz keine Möglichkeit zum dynamischen Anlegen von Detailelementen gegeben.

10.1.2.5 Export der SQL-MDi Datei

Der Export der SQL-MDi Datei erfolgt durch eine Exportfunktion, welche über das Kontextmenü des Projekts aufgerufen werden kann: *Export → Other → Create Sql-MDi File*. Im Exportassistenten muss der Dateiname angegeben werden und die Datei wird vom System erstellt. Sollten Fehler in der SQL-MDi Datei auftreten, wird eine Fehlermeldung angezeigt und eine *.log*-Datei mit den detaillierten Fehlermeldungen erzeugt.

Nach Erstellen der Datei wird diese noch nicht im Projekt direkt angezeigt, dazu muss der Projektordner aktualisiert werden. Hierzu muss die Taste *F5* beim ausgewählten Projekt gedrückt werden. Die SQL-MDi Datei zum verwendeten Rahmenbeispiel ist in Abbildung 10-14 dargestellt.




```
DEFINE CUBE dw1::verkauf AS c1
(MEASURE c1.dauer, MEASURE c1.umsatz_telefonie, MEASURE c1.umsatz_sonstiges, Dim c1.datum_std , Dim c1.svnr , Dim c1
PIVOT MEASURES c1.umsatz_telefonie, c1.umsatz_sonstiges INTO c1.umsatz USING c1.umsatzkategorie)
 (ROLLUP c1.datum_std TO LEVEL c1.datum[datum] )

 CUBE dw2::verkauf AS c2
(MEASURE c2.gespraechsdauer -> dauer, MEASURE c2.umsatz, Dim c2.datum  (MAP LEVELS c2.datum ([datum], [month -> monat

 (CONVERT MEASURES APPLY usd2Eur() FOR c2.umsatz DEFAULT)


 GLOBAL CUBE MAISLINGER::verkauf AS c0

 MERGE DIMENSIONS c1.datum AS d1, c2.datum AS d2 INTO c0.datum AS d0

 MERGE DIMENSIONS c1.kunde AS d3, c2.kunde AS d4 INTO c0.kunde AS d5
(RELATE d3.svnr,d4.kunde WHERE d3.svnr=d4.kunde USING HIERARCHY OF d3)
(MATCH ATTRIBUTES d3.bezeichnung IS d4.name)
(CONVERT ATTRIBUTES APPLY usd2Eur() FOR d4.grundgebuehr DEFAULT)

 MERGE DIMENSIONS c1.mobilnetz AS d6, c2.mobilnetz AS d7 INTO c0.mobilnetz AS d8
(RENAME d6.mobilnetz >> 'HandyTelInc' WHERE c1.mobilnetz='HandyTel')

 MERGE DIMENSIONS c1.umsatzkategorie AS dp33, c2.umsatzkategorie AS d10 INTO c0.umsatzkategorie AS d11

 MERGE CUBES c1, c2 INTO c0 ON datum, svnr, mobilnetz, umsatzkategorie
AGGREGATE MEASURE umsatz IS SUM OF umsatz,
AGGREGATE MEASURE dauer IS SUM OF dauer
(MEASURE umsatz,MEASURE dauer, DIM datum, DIM svnr, DIM mobilnetz, DIM umsatzkategorie)
```

Abbildung 10-14: Erstellte SQL-MDi Datei

10.1.2.6 Export der Metadaten

Der Export der Metadaten ist notwendig, um Kompatibilität mit dem SQL-MDi Query Parser und dem SQL-MDi Query Processor zu erreichen. Die Ausführung des dazugehörigen Assistenten erfolgt durch den Aufruf von *Export → Other → Export Metadata to Database*. Im Assistenten müssen lediglich die Zugangsdaten zu dem bei der Installation angegebenen *OLAP_Repository* eingegeben werden. Nach Betätigung der *Finish*-Schaltfläche wird der Exportvorgang eingeleitet. Dieser Vorgang kann einige Sekunden bis Minuten dauern. Der Fortschritt wird währenddessen durch einen Statusbalken angezeigt.

10.1.2.7 Symbole und Schaltflächen

Im Global Schema Architect werden Symbole zur Darstellung von Elemente eingesetzt. In folgender Tabelle werden die verschiedenen Symbole sowie deren Verwendungszweck nochmals zusammengefasst.

Symbol	Verwendungszweck
	Darstellung eines Würfels (engl. Cube)
	Darstellung einer Dimension (engl. Dimension)
	Darstellung einer Aggregationsebene (engl. Level)
	Darstellung eines Fakts (engl. Fact)
	Darstellung einer Kenngröße (engl. Measure)
	Allgemeines Symbol für Global Schema Architect
	Erzeugen einer neuen Datei
	Export der Metadaten
	Erzeugen und Exportieren von Mappings

Tabelle 10-1: Verwendungszweck von Symbolen

10.1.3 Zusammenfassung

Im Benutzerhandbuch wird die Installation des Global Schema Architects ausführlich beschrieben. Darauffolgend wurden die verschiedenen Anwendungsmöglichkeiten im Detail aufgezeigt. Um eine möglichst einfache Erklärung zum besseren Verständnis bereitzustellen, wurden Screenshots und Aktivitätsdiagramme eingesetzt.

Es wurde auf eine genaue Beschreibung der verschiedenen SQL-MDi Elemente verzichtet, da für diese ein Grundwissen der SQL-MDi notwendig ist. Für die Beschreibung dieser Technologie wird auf [Berger & Schrefl 2006] verwiesen. Durch den intuitiven Aufbau der Mapping-Editoren ist die Anwendung dieses Wissens einfach anwendbar.

Schlussendlich wurden die wichtigsten Symbole des Global Schema Architects nochmals dargestellt und beschrieben.

11 Anhang B

11.1 Abbildungsverzeichnis

11.2 Tabellenverzeichnis

11.3 Listingverzeichnis

11.4 Definitionsverzeichnis

11.5 Literaturverzeichnis

[Abelló et al. 2006] Abelló A.; Lechtenbörger J.; Rizzi S.; Trujillo J.: *Research in Data Warehouse Modeling and Design: Dead or Alive?* In: Sogn I.-Y.; Vasiliadis P (Hrsg.), DOLAP, Seiten 3 – 10, ACM, 2006.

[Akoka et al. 2006] Akoka J.; Prat N.; Comyn-Wattiau I.: *A UML-based data warehouse design method.* In: Decision Support Systems 42, Seiten 1449 – 1473, 2006.

[Bauer & Günzel 2004] Bauer A.; Günzel H.(Hrsg.): *Data-Warehouse-Systeme Architektur – Entwicklung – Anwendung*. 2. Überarbeitete und aktualisierte Auflage, dpunkt.verlag, Heidelberg, 2004.

[Berger & Schrefl 2006] Berger S.; Schrefl M.: *Analysing Mulit-dimensional Data Across Autonomous Data Warehouses*. In: DaWak, Seiten 120 – 133, 2006.

[Berger & Schrefl 2008] Berger S.; Schrefl M.: *From Federated Databases to a Federated Data Warehouse System*. IEEE - Proceedings of the 41st Hawaii International Conference on System Sciences (HICSS-41 2008), 2008.

[Betsy et al. 2009] Betsy B.; Serface L.; Wong R.: *UI Models–Master Detail Templates. Oracle Corporation* http://www.oracle.com/technology/tech/blaf/specs/masterDetail_template.html (letzter Abruf 10. April 2009)

[Budinsky et al. 2004] Budinsky F.; Steinberg D.; Merks E.; Ellersick R.; Grose T. J.: *Eclipse Modelling Framework – A Developer's Guide*. Addison-Wesley, 2004.

[Bruck & Hussey 2008] Bruck J.; Hussey K.: *Customizing UML: Which Technique is Right for You?* http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing_UML2_Which_Technique_is_Right_For_You/article.html#_Hlt174419544 (letzter Abruf 05. Februar 2009)

[Bruck & Damus 2008] Bruck J.; Hussey K.: *Creating Robust Scalable DSLs with UML* http://www.eclipse.org/modeling/mdt/uml2/docs/tutorials/EclipseCon2008_Tutorial_Creating_Robust_Scalable_DSL_with_UML_files/frame.html (letzter Abruf 05. Februar 2009)

[Brunneder 2008] Brunneder W.: *Entwicklung eines Parser für SQL-MDi, eine multidimensional Abfragesprache für Föderierte Data Warehouse-Systeme*. Linz, Österreich, 2008.

[Cabibbo & Torlone 2004] Cabibbo L.; Torlone R.: *On the Integration of Autonomous Data Marts.* Proceedings of the 16th International Conference on Scientific and Statistical Database Mangagement (SSDBM'04), IEEE, Seiten 223-224, 2004.

[Chang et al. 2002] Poole J.; Chang D.; Tolbert D.; Mellor D.: *Common Warehouse Metamodel*. New York, USA, John Wiley and Sons Inc., 2002.

[Chamoni & Gluchowski 1998] Chamoni P.; Gluchowski P.: *Analytische Informationssysteme – Einordnung und Überblick*. In: Chamoni P.; Gluchowski P. (Hrsg.): Analytische Informationssysteme. Berlin, Deutschland, S. 27-45, Springer, 1998.

[Chaudhury & Dayal 1997] Chaudhuri S.; Dayal U.: *An overview of data warehouseing and OLAP technology.* In: ACM SIGMOD Record Vol. 26, Seiten. 65 – 74, ACM, 1997.

[Codd et al. 1993] Codd E.; Codd S.; Salley C.: *Beyond Decision Support*. In: Computerworld 27, 1993, S. 87-89.

[Clark & Derose 1999] Clark J.; Derose S.: *XML Path Language (XPath)*. http://www.w3.org/TR/xpath (letzter Abruf am 08. Februar 2009)

[Dadam 1996] Dadam P.: *Verteilte Datenbanken und Client/server-systene: Grundlagen Konzepte und Realisierungsformen*. Springer, 1996.

[Daum 2005] Daum B.: *Rich-Client-Entwicklung mit Eclipse 3.1- Anwendungen entwickeln mit der Rich Client Platform.* dpunkt.verlag, Heidelberg, 2005.

[Dresden OCL 2009] Dresden OCL Toolkit http://dresden-ocl.sourceforge.net/ (letzter Abruf 16. April 2009)

[Eclipse EPL-FAQ 2009] Eclipse Public License (EPL): *Frequently Asked Questions.* http://www.eclipse.org/legal/eplfaq.php (letzter Abruf 11. Jänner 2009)

[Eclipse GEF 2009] Graphical Editor Framework (GEF)
http://www.eclipse.org/gef/ (letzter Abruf 20. Februar 2009)

[Eclipse GEF FAQ 2009] Graphical Editor Framework: *Frequently Asked Questions.* http://wiki.eclipse.org/GEF_Developer_FAQ (letzter Abruf 20. Februar 2009)

[Eclipse Org 2009] Eclipse.org: *About the Eclipse Foundation*.
http://www.eclipse.org./org/ (letzter Abruf 11.Jänner 2009)

[Eclipse Org-PR 2009] Eclipse Forms Independent Organization (Press Relase)
http://www.eclipse.org/org/press-release/feb2004foundationpr.php/ (letzter Abruf 11. Jänner 2009)

[Eclipse Tools 2009] Eclipse.org: Eclipse Tools Project.
http://www.eclipse.org./tools/ (letzter Abruf 04. Jänner 2009)

[Eclipse UML2 2009] MDT/UML
http://wiki.eclipse.org/MDT-UML2 (letzter Abruf 20. Februar 2009)

[Eclipse UML2 Co. 2009] MDT-UML2-Tool-Compatibility
http://wiki.eclipse.org/MDT-UML2-Tool-Compatibility (letzter Abruf 20. Februar 2009)

[Eclipse UML2 FAQ 2009] MDT/UML/FAQ
http://wiki.eclipse.org/MDT/UML2/FAQ (letzter Abruf 20. Februar 2009)

[Eclipse UML2 Tools 2009] MDT-UML2Tools
http://wiki.eclipse.org/MDT-UML2Tools (letzter Abruf 20. Februar 2009)

[Eclipse UML2 T. FAQ 2009] MDT-UML2Tools FAQ http://wiki.eclipse.org/MDT-UML2Tools_FAQ (letzter Abruf 20. Februar 2009)

[Gardner 1998] Gardner S. R.: *Building the data warehouse*. In: Communications of the ACM, Vol. 41, Seiten 52 – 60, ACM, 1998.

[Gamma et al. 2002] Gamma E.; Helm R.; Johnson R.; Vlissides J.: *Design Patterns – Elements of Reusable Object-Oriented Software*. 24. Auflage, Boston, Addison-Wesley, 2002.

[Gamma et al. 2006] Gamma E.; Nackman L.; Wiegand J: *eclipse – Building Commercial-Quality Plug-ins*. 2. Auflage, Stoughton, Addison-Wesley, 2006.

[GMF 2009] Graphical Modeling Framework http://wiki.eclipse.org/index.php/Graphical_Modeling_Framework (letzter Abruf 05. Februar 2009)

[Glozic 2005] Eclipse Forms: Rich UI for the Rich Client http://www.eclipse.org/articles/Article-Forms/article.html (letzter Abruf 06. Februar 2009)

[Golfarelli et al. 1998] Golfarelli M.; Maio D.; Rizzi S.: *The Dimensional Fact Model: A Conceptual Model For Data Warehouses*. In: international Journal of Cooperative Information Systems 7, Seiten 215-247, 1998.

[Golfarelli & Rizzi 1998] Golfarelli M.; Rizzi S.: *A methodological framework for data warehouse design*. In: Proceedings of the 1st ACM international workshop on Data warehousing and OLAP. New York, ACM, Seiten 3 – 9, 1998.

[Gruhn et al. 2006] Gruhn V.; Pieper D.; Röttgers C: *MDA – Effektives Software-Engineering mit UML 2 und Eclipse*. Springer-Verlag, Heidelberg, 2006.

[Heinrich et al. 2004] Heinrich, J. L.; Heinzl, A.; Roithmayr, F.: *Wirtschaftsinformatik-Lexikon*. 7. Auflage, Oldenburg Verlag, München/Wien, 2004.

[Hüsemann et al. 2000] Hüsemann B.; Lechtenbörger J.; Vossen G.: *Conceptual data warehouse design*. In: Proc. DMDW, Seiten 3 – 9, 2000.

[Inmon 1996] Inmon W.H.: *Building the Data Warehouse*. Wiley Publishing, Inc., 2th Edition, 1996.

[JDBC 2009] *The Java Database Connectivity (JDBC)* http://java.sun.com/javase/technologies/database/ (letzter Abruf am 08. Februar 2009)

[Karagiannis & Kühn 2002] Karagiannis d.; Kühn H.: *Metamodelling Platforms*. In: Bauknecht k.; Min Tjoa A.; Quirchmayer G. (Hrsg.): Proceedings of the Third International Conference EC-Web 2002 – Dexa 2002, Aix-en-Provence, France, Springer-Verlag, Berling, Heidelberg, 2002.

[Kemper & Eickler 2006] Kemper A.; Eickler A.: *Datenbanksysteme: Eine Einführung*. 6. Auflage. Oldenbourg Wissenschaftsverlag, 2006.

[Luján-Mora et al. 2002] Luján-Mora S.; Trujillo J.; Song I.: *Multidimensional Modelling with UML Package Diagrams*. In: Conceptual Modelling – ER 2002, Springer-Verlag Berlin, Seiten 199 – 213, 2002.

[Luján-Mora & Trujillo 2003] Luján-Mora S.; Trujillo J.: *A comprehensive method for data warehouse design*. In: Proc. DMDW, 2003.

[OMG 2000] Object Management Group
http://www.jeckle.de/files/Infrastructure00-09-01.pdf (letzter Abruf 07.03.2009)

[OMG CORBBA 2009] Common Object Request Broker Architecture
http://www.corba.org/ (letzter Abruf 16.04.2009)

[OMG CWM 2009] Documents Associated With Common Warehouse Metamodel.
http://www.omg.org/spec/CWM/1.1/ (letzter Abruf 16.04.2009)

[OMG MDA 2009] Model Driven Architecture
http://www.omg.org/mda/ (letzter Abruf 16.04.2009)

[OMG UML 2009] UML Resource Page
http://www.uml.org/ (letzter Abruf 16.04.2009)

[OMG XMI 2009] CORBA, XML and XMI Resource Page
http://www.omg.org/xml/ (letzter Abruf 16.04.2009)

[Pendse & Creeth 1995] Pendse N.; Creeth R.: *The OLAP Report Succeeding with Online Analyical Processing*. Norwalk, CT: Business Intelligence Inc, 1995.

[Pomberger & Blaschek 1996] Pomberger G.; Blaschek G.: *Object-orientation and prototyping in software engineering*. 3. Auflage, Carl Manser Verlag, München und Wien, 2004.

[Pomberger & Pree 2004] Pomberger G.; Pree W.: *Software Enginieering- Architektur-Design und Prozessorientierung*. Carl Manser Verlag, München und Wien, 1996.

[Rossgatterer 2008] Rossgatterer T.: *Entwicklung eines Query Prozessors in einem Föderierten Data Warehouse System*. Linz, Österreich, 2008.

[Schäling 2005] Schäling B.: *Der moderne Softwareentwicklungsprozess mit UML*. http://www.highscore.de/uml/ (letzter Abruf 03. Februar 2009)

[Sheth & Larson 1990] Sheth P.; Larson A.: *Federated Database Systems for Managing Distributed Hetergenous, and Autonomous Databases*. In: ACM Cumputing Surveys, Vol. 22, Nr. 3, Seiten 183 - 36, 1990.

[SWT 2009] SWT: The Standard Widget Tookit. http://www.eclipse.org/swt/ (letzter Abruf am 08.Februar 2009)

[Torlone 2008] Torlone R.: *Two approaches to the integration of heterogeneous data warehouses*. In: Distributed and Parallel Databases, Vol. 23, Seiten 69 – 97, Springer Netherlands, 2008.

[Uckat 2007] Uckat B.: *Metamodellierung mit MOF und Ecore und deren Anwendung im Rahmen des MDA-Ansatzes*. In: Ausgewählte Themen des Software Engineering. http://www.wi.uni-muenster.de/pi/lehre/ss07/SeminarSE/vortraege/SeminarSE_Praesentation_Uckat.pdf (letzter Abruf 07.03.2009)

[UML 2.0 2003] *UML 2.0 Infrastructure Specification* http://www.jeckle.de/files/03-09-15.pdf (letzter Abruf 08. Februar 2009)

[Ullenboom 2007] Ullenboom C: *Java ist auch eine Insel – Programmieren mit der Java Standard Edition Version 6*. 7. Auflage, Galileo Computing, 2007.
http://openbook.galileocomputing.de/javainsel7/ (letzter Abruf 08. Februar 2009)